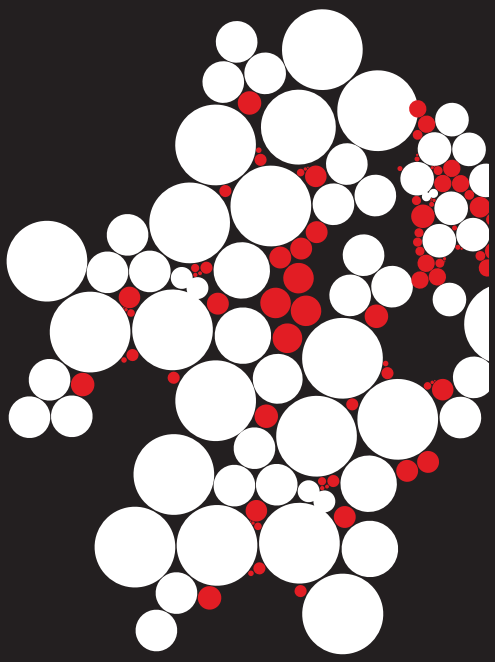
MASTER THESIS



BUSINESS PROCESS MANAGEMENT IN THE CLOUD WITH DATA AND ACTIVITY DISTRIBUTION

Evert F. Duipmans

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE
SOFTWARE ENGINEERING

EXAMINATION COMMITTEE
dr. Luís Ferreira Pires
dr. Ivan Kurtev
dr. Luiz O. Bonino da Silva Santos
dr. Dick A. C. Quartel

DOCUMENT NUMBER
EWI/SE - 2012-002

UNIVERSITY OF TWENTE.

SEPTEMBER 2012

Abstract

Business Process Management (BPM) gives organizations the ability to identify, monitor and optimize their business processes. A Business Process Management System (BPMS) is used to keep track of business process models and to coordinate the execution of business processes.

Organizations that want to make use of BPM might have to deal with high upfront investments, since not only software and hardware needs to be purchased, but also personnel needs to be hired for installing and maintaining the systems. In addition, scalability can be a concern for an organization. A BPMS can only coordinate a certain amount of business process instances simultaneously. In order to serve all customers in peak-load situations, additional machines are necessary. Especially when these machines are only rarely needed, buying and maintaining the machines might become expensive.

Nowadays, many BPM vendors offer cloud-based BPM systems. The advantage of these systems is that organizations can use BPM software in a pay-per-use manner. In addition, the cloud solution should offer scalability to the user, so that in peak-load situations, additional resources can be instantiated relatively easily. A major concern of cloud-based solutions for organizations, is the fear of losing or exposing confidential data. Since the cloud solution is hosted outside an organization and data is stored within the cloud, organizations fear they might lose control over their data.

In this report we consider a BPM architecture in which parts of a business process are placed in the cloud and parts are placed on-premise. A decomposition framework for automatically decomposing a business process into collaborating business processes for deployment in the cloud or on-premise is developed in this work. The decomposition is driven by a list of markings in which the distribution location for each of the activities in a business process is defined. In addition, data restrictions can be defined, to ensure that sensitive data does not cross the organizational borders.

The solution we present is business process language independent. An intermediate model is used for capturing the behavior of a business process and the decomposition is performed on this intermediate model, rather than on an existing language. The decomposition framework consists of a transformation chain of three transformations, used for converting a business process, defined in an existing business process language into an instance of the intermediate model, performing the decomposition algorithm on the instance and transforming the result back into a business process defined in an existing business process language.

We analyze possible transformation rules, explain the implementation of the transformations and show an example of the decomposition framework by performing a case study.

Preface

This thesis presents the results of the final assignment in order to obtain the degree Master of Science. The project was carried out at BiZZdesign in Enschede. Finishing university is a an important achievement in my life. Therefore, I would like to thank a couple of people:

First, I would like to thank Luís for being my first supervisor on behalf of the university. Thanks for the great meetings and discussions we had. Your (red) comments helped me to really improve the report.

Second, I want to thank Ivan, my second supervisor on behalf of the university. Although we did not talk much about the subject of my thesis, I appreciate the conversations we had about computer science, life and music. I wish you all the best in finding a new university to continue your research.

Third, I would like to thank Luiz and Dick for supervising on behalf of BiZZdesign. Thanks for the meetings we had during the last 6 months. The discussions we had helped me to stay critical and obtain this result.

Fourth, I want to thank BiZZdesign for letting me perform my master assignment at their office. I want to thank all the colleagues and fellow students in the company for their support and the great time we had.

When I moved to Enschede three years ago, I barely knew anyone here. The last three years I have met many people and I have been encouraged to try new (especially cultural) activities. I want to thank my friends, my house mates and the other people I met the last three years. Thanks for all the time we spend together and for helping me to have a fantastic time in Enschede.

Finally, I want to thank my sister Amarins, my brother Gerard, and the most important people in my life: my parents. Your love and support have helped me not only through university, but also through life. Thanks for inspiring me to get the most out of life and supporting me in every decision I have made.

Evert Ferdinand Duipmans

Enschede, September 2012

Contents

1. Introduction

The structure of this chapter is as follows: Section 1.1 presents the motivation for this work. Section 1.2 defines the research objectives. Section 1.3 describes the approach that was followed to achieve the objectives. Section 1.4 presents the structure of this report.

1.1 Motivation

Business Process Management (BPM) has gained a lot of popularity the last two decades. By applying BPM, organizations are able to identify, monitor and optimize their business processes. This may lead to lower costs, better customer satisfaction or optimized processes for creating new products at lower cost [1].

A business process can be described by a workflow, consisting of activities. The activities in a process can be either human-based, system-based or a combination of both, in case of human-system interaction. A Business Process Management System (BPMS) is often used for coordinating the execution of a business process. The system manages business process models and keeps track of running instances of the process models. A workflow engine is used for the execution of the process models. The BPMS is equipped with a monitoring tool, to give organizations the opportunity to get insight into running and finished processes.

Organizations that want to embrace BPM technology might face high upfront investments, since not only software, but also hardware needs to be purchased. In addition, personnel needs to be hired for setting up and maintaining the system. Scalability can also be a concern for companies that use BPM, since a process engine is only able to coordinate a limited number of business process instances simultaneously. Organizations might need to purchase additional machines to ensure that their customers can still be served during peak load situations. When these situations only occur rarely, the additional machines might make BPM expensive, since the fixed costs for maintenance still have to be paid by the organization.

Cloud computing [2] gives users the opportunity of using computing resources in a pay-per-use manner and perceiving these resources as being unlimited. The cloud computing definition by the NIST [3] mentions three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Several software vendors offer cloud-based BPM solutions. Instead of having to make upfront investments, organizations can use BPM software in a pay-per-use manner. A cloud-based solution should also be scalable, which gives organizations the freedom to scale up and down relatively easily according to their needs.

A major concern for organizations is often the security of cloud-based solutions. Organizations fear that they might lose or expose sensitive data, by placing these data in the cloud. In addition, not all activities might benefit from being placed in the cloud. For example, the execution of non-computation-intensive activities might be as fast as in the cloud, or even faster, if large amounts of data need to be sent to the cloud first, in order to execute the activity. This might also make a cloud solution expensive, since data transfer is commonly one of the billing factors of cloud computing.

The idea of splitting up a business process and placing activities and data in both the organization and the cloud has been proposed in [4]. The paper describes and architecture in which organizations can place their sensitive data and non-computation-intensive activities within the organization itself, whereas less sensitive data and scalable activities can be placed in the cloud. Decomposition of the original monolithic process into separate individual processes is however not addressed in [4].

1.2 Objectives

In this research we work towards an architecture, based on [4], in which two process engines, one placed on-premise and one placed in the cloud, are used for coordinating a business process. We split-up the original process and distribute it to both engines based on location assignments for activities and data, to benefit from the advantages of both traditional and cloud-based BPM.

The main research objective of this thesis is:

> *Business Process Management in the cloud with protection of sensitive data and distribution of activities through the decomposition of monolithic business processes into individual processes.*

To achieve this goal, we developed a decomposition framework which is able to transform a business process into several separate processes, based on activities that are marked with a distribution location. During the development of the framework, the following research questions are considered:

RQ1. Which approaches are available for decomposing a business process?
Much research has been performed on the decentralization of orchestrations. We identified a base language to define our business process and we selected a technique for specifying the process transformation.

RQ2. Which transformation rules can be applied to constructs within a business process?

We performed an analysis to identify possible transformation rules that are applicable to our business processes, when nodes within the process are marked with a distribution location.

RQ3. How to deal with data restrictions when decomposing a business process?
We introduced data dependencies between the activities in a business process by performing data analysis and introduced a data restriction verification algorithm to ensure that no data restrictions are violated during the decomposition transformation.

RQ4. How to verify the correctness of the decomposition solution?
We verified the correctness of the decomposition solution informally by showing that the resulting business processes can be transformed back to the original business process by merging the obtained business processes and removing the communication nodes.

1.3 Approach

To achieve the main objective of this research and answer the research questions, the following steps have been taken:

1. Perform a literature study on BPM and cloud computing, by looking at both subjects individually, but by also investigating approaches in which BPM and cloud computing are combined.
2. Survey literature on the decomposition of business processes to find a representation for business processes that captures their structure and semantics.
3. Define and implement transformation rules to enable decomposition of business processes.
4. Test the framework by performing a case study.
5. Verify that activities are correctly distributed and no data restrictions are violated.

1.4 Structure

The remainder of this thesis is structured as follows.

Chapter 2 gives the background for our work. We introduce and discuss both Business Process Management and Cloud Computing in some detail.

Chapter 3 defines the approach we use for the decomposition and discusses related work.

Chapter 4 introduces the intermediate model we use for the decomposition transformation.

Chapter 5 analyzes possible transformation rules for the decomposition phase.

Chapter 6 reports on the implementation of the decomposition transformation. Each of the phases of the algorithm is explained in pseudo code.

Chapter 7 discusses the business process language we have selected for our lifting and grounding transformation. The language is explained and a mapping between the business process language and the intermediate model is investigated.

Chapter 8 discusses our lifting and grounding transformations. Each of the phases of the transformations is explained with code examples.

Chapter 9 gives an example of the transformation chain, by performing the transformations on a case study.

Chapter 10 concludes this report and gives recommendations for further research.

2. Background

This chapter is structured as follows: Section 2.1 introduces Business Process Management by explaining the BPM lifecycle, orchestrations and choreographies and the structure of a BPMS. Section 2.2 describes cloud computing, by investigating the service models and cloud types. The benefits and drawbacks for both cloud computing in general and each of the service models are identified. Section 2.3 introduces BPM in the cloud by looking at specific benefits and drawbacks and by introducing an architecture in which traditional BPM and cloud-based BPM are combined.

2.1 Business Process Management

The goal of BPM is to identify the internal business processes of an organization, capture these processes in process models, manage and optimize these processes by monitoring and reviewing them.

Business process management is based on the observation that each product that a company provides to the market is the outcome of a number of activities performed [1]. These activities can be performed by humans, systems or a combination of both. By identifying and structuring these activities in workflows, companies get insight into their business processes. By monitoring and reviewing their processes, companies are able to identify the problems within these processes and can come up with improvements.

2.1.1 BPM lifecycle

The BPM lifecycle is an iterative process in which all of the BPM aspects are covered. A simplified version of the BPM lifecyle is shown in Figure 2.1. Below we briefly introduce each of the phases of the BPM lifecycle.

Design

In the design phase the business processes within a company are identified. The goal of the design phase is to capture the processes in business process models. These models are often defined using a graphical notation. In this way, stakeholders are able to understand the process and refine the models relatively easily. The activities within a process are identified by surveying the already existing business process, by considering the structure of the organization and by identifying the technical resources within the company. BPMN

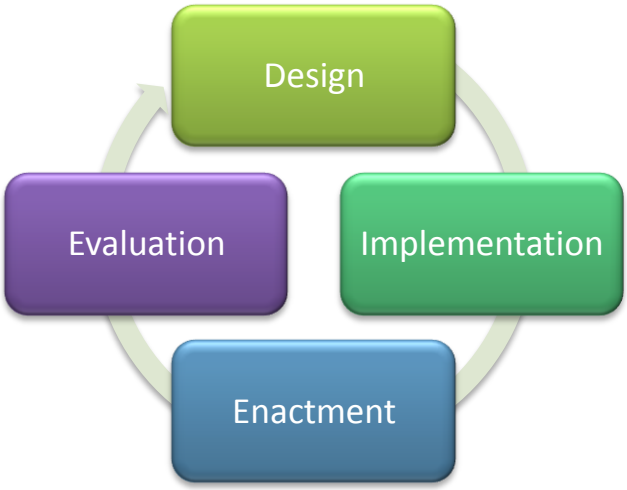


Figure 2.1: Schematic representation of the business process management lifecycle [1]

[5] is the most popular graphical language for capturing business process models in the design phase.

When the business processes are captured within models, these models can be simulated and validated. By validating and simulating the process, the stakeholders get insight into the correctness and suitability of the business process models.

Implementation

After the business process models are validated and simulated, they have to be implemented. The implementation of these models can be done in two ways:

1. One can choose to create work lists, with well defined tasks, which can then be assigned to workers within the company. This is often the case when no automation is necessary or possible within the business process execution. The disadvantage of working with work lists is that the process execution is hard to monitor. There is no central system in which process instances are monitored, and this has to be done by each employee within the company who is involved in the process.

2. In a lot of situations information systems participate in a business process, in which case a business process management system (BPMS) can be used. A BPMS is able to use business process models and create instances of these models for each process initiation. The advantage of using a BPMS is that the system gives insight into the whole process. The system is able to monitor each instance of a business process and gives an overview of the activities that are performed, the time the process takes and its completion or failure.

Business Process Management Systems need executable business models. The models defined in the design phase are often too abstract to be directly executed. Therefore, they need to be implemented in an executable business process language, such as BPEL [6]. In addition, collaborations between business processes can be described by using a choreography language, such as CDL [7].

Enactment

When the business process models are implemented in the implementation phase, the enactment phase can be started. In this phase the system is used at runtime, so that each initiation of the process is monitored and coordinated by the BPMS. For each initiation of a process, a process instance is created. The BPMS keeps track of the progress within each of the process instances. The most important tool within the enactment phase is the monitoring tool, since it gives an overview of the running and finished process instances. By keeping track of these instances, problems that occur in a process instance can be easily detected.

Evaluation

In the evaluation phase the monitored information that is collected by the BPMS is used to review the business process. The conclusions drawn in the evaluation phase are used as input for the next iteration of the lifecycle.

2.1.2 Orchestration vs. Choreography

An Orchestration describes how services can interact with each other at the message level, including the business logic and execution order of the interactions from the perspective and under control of single endpoint [8].

A choreography is typically associated with the public message exchanges, rules of interaction, and agreements that occur between multiple business process endpoints, rather than a specific business process that is executed by a single party [8]. An example of a language for defining choreographies is CDL [7]. CDL allows its users to describe how peer-to-peer participants communicate within the choreography. Choreography specifications give organizations the opportunity to collaborate, using a collaborative contract. The interaction between partners is clearly defined, but the implementation of the individual orchestrations is the responsibility of each of the participants.

2.1.3 Business Process Management System (BPMS)

Several vendors of Business Process Management software solutions offer complete suites for modeling, managing and monitoring business processes. Inside these systems there is a process execution environment, which is responsible for the enactment phase of the BPM lifecycle [1]. An abstract schema of a typical BPMS is shown in Figure 2.2.

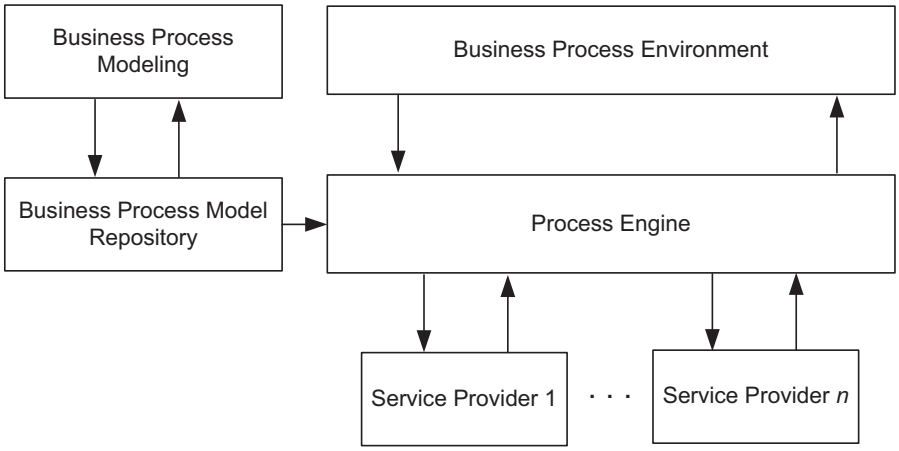


Figure 2.2: Schematic representation of a business process management system [1]

The tools shown in Figure 2.2 provide the following functionality:

- The Business Process Modeling component consists of tools for creating business process models. It often consists of graphical tools for developing the models.
- Business Process Environment is the main component that triggers the instantiation of process models.
- The Business Process Model Repository is a storage facility for storing process models as created by the modeling component.
- The Process Engine keeps track of the running instances of process models. It communicates with service providers in order to execute activities or receive status updates.
- Service Providers are the information systems or humans that communicate with the process engine. These entities perform the actual activities and report to the process engine.

2.2 Cloud Computing

Cloud computing is one of the trending topics in Computer Science nowadays. Many market influencing players as Microsoft, Google and Amazon offer cloud computing solutions. The goal of this section is to introduce cloud computing from both a conceptual level and a more concrete level. At first the general benefits and drawbacks of cloud computing are explained briefly. The three common service models are introduced next and for each of these service models its specific benefits and drawbacks are identified. After that, four different cloud types are discussed.

2.2.1 General benefits and drawbacks

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [3].

The idea of cloud computing is that users are offered computing resources in a pay-per-use manner that are perceived as being unlimited. The cloud provider does not have any expectations or up-front commitments with the user and it offers the user elasticity to quickly scale up or down according to the user's needs.

Cloud computing gives organizations several benefits:

- **Elasticity**
  Instead of having to buy additional machines, computing resources can be reserved and released as needed. This means that there is no under- or over-provisioning of hardware by the cloud user.
- **Pay-per-use**
  Cloud users are only billed for the resources they use. If a cloud user needs 20 computers once a week for some computation of one hour, it is only billed for these computing hours. After that the computers can be released and can be used by other cloud users.
- **No hardware maintenance**
  The computing resources are maintained by the cloud provider. This means that operational issues such as data redundancy and hardware maintenance are attended by the cloud provider instead of the cloud user.
- **Availability**
  Clouds are accessible over the Internet. This gives cloud users the flexibility to access their resources over the Internet. Cloud users are able to use software or data that

is stored in the cloud not only inside their organization but everywhere they are provided with Internet access.

There are also drawbacks and threats in using cloud computing:

- **Security**
  Data is stored inside the cloud and accessible through the Internet. In several situations cloud users deal with confidential information that should be kept inside the cloud user's organization. In these situations cloud computing might not be a good solution, although there are solutions with cloud computing in which data is stored inside the cloud user's organization but applications are hosted in the cloud. There are also technical solutions for making data unintelligible for unauthorized people, for example, by using encryption algorithms.

- **Availability**
  Clouds are accessible through the Internet. This gives cloud users the freedom to work with the services wherever they have an Internet connection. The downside is that when the Internet connection fails, for example, on the side of the cloud provider, cloud users are not able to access their services any more. This might lead to business failures, especially when the services are part of a business process.

- **Data transfer bottlenecks**
  Users that use software systems might need to transfer large amounts of data in order to use the system. Data should be transported not only from the user to the system, but also to multiple systems in order to cooperate inside a company. Cloud computing providers do not only bill the computation and storage services, but also data transportation is measured and billed. For companies that deal with a lot of data, cloud computing may be expensive because of the data transportation costs. Another problem can be the time it takes to transfer data to the cloud. For example, suppose that a company needs to upload a huge amount of data in order to perform a complex computation, in which case the data transfer may take more time than the computation itself. In these situations it might be faster and cheaper to perform the computation inside the premises of the cloud user.

2.2.2 Service models

The National Institute of Standards and Technology (NIST) identifies three service models for cloud computing: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) [3]. The three service models are closely related and can be seen as a layered architecture, as shown in Figure 2.3. Each service model is explained in the sequel. For each of the models, its specific benefits and drawbacks are discussed for

both the user and the provider of the service models.

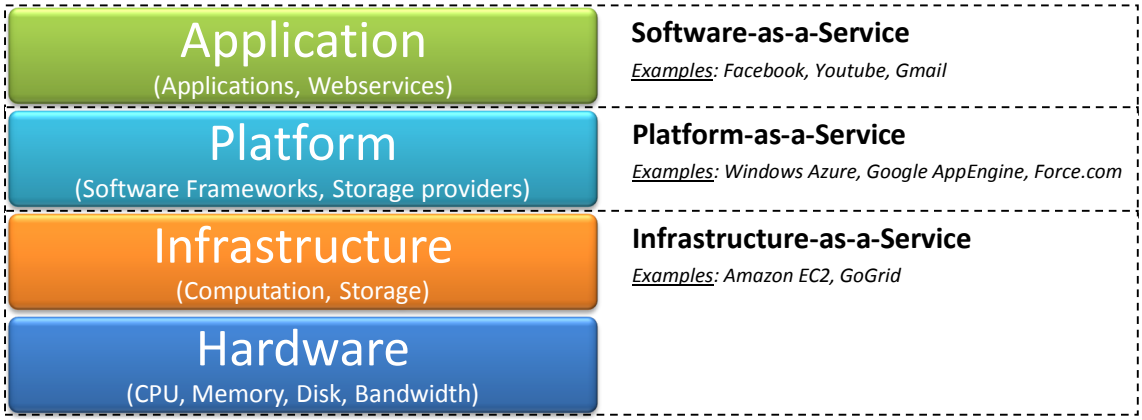


Figure 2.3: An overview of the layers in cloud computing based on [9]

Infrastructure-as-a-Service (IaaS)

Infrastructure-as-a-Service is the lowest layer in the cloud computing stack. As shown in Figure 2.3, IaaS combines two layers: the hardware layer and the infrastructure layer. IaaS users are interested in using hardware resources such as CPU power or disk storage. Instead of directly offering these services to the user, IaaS providers provide users with a virtualization platform. Customers need to install and configure a virtual machine, which runs on the hardware of the cloud provider. In this model, cloud users are responsible for their virtual machine and cloud providers are responsible for the actual hardware. Issues such as data replication and hardware maintenance are addressed by the cloud provider, while the management of the virtual machine is performed by the cloud user.

Benefits of IaaS for cloud users are:

- **Scalable infrastructure**
  The biggest advantage of IaaS is the elasticity of the service. Instead of having to buy servers, software and data center capabilities, users rent these resources on a pay-per-use manner. In situations where the workload of computer resources fluctuates, IaaS might be a good solution. For example, consider a movie company that uses servers for rendering 3D effects. The company has a small data center on-premise which is used for the rendering, once a week. The rendering of one scene takes 50 hours when performed on 1 machine. By scaling up to 50 machines, the rendering of the scene would take 1 hour. Scaling up the internal network of the company might be an expensive operation considering the installation and maintenance of the machines, especially when the servers are only used for rendering once a week.

Instead of buying these machines, one might consider to rent the machines and only pay for the rendering operation once a week.

- **Portability**
  Since IaaS works with virtual machine images, porting an on-premise system to the cloud or porting a virtual machine from one cloud to another can be relatively easy. This, however, depends on the virtual machine image format that is used by the cloud provider.

Drawbacks of IaaS for cloud users are:

- **Virtual machine management**
  Although cloud users do not have to manage the rented computer hardware, cloud users are still responsible for the installation and configuration of their virtual machine. A cloud user still needs experts inside its organization for the management of these virtual servers.

- **Manual scalability**
  IaaS does no offer automated scalability to applications. Users are able to run virtual machines and might boot several instances of virtual machines in order to scale up to their needs. Collaboration between the virtual machines has to be coordinated and programmed by the cloud user.

Benefits for IaaS cloud providers are:

- **Focus on hardware**
  Cloud providers are mainly focused on hardware related issues. Everything that is software related, such as database management, threading and caching needs to be performed by the cloud user.

- **Exploiting internal structure**
  Several providers are able to offer cloud computing services as an extension to their core business. For example, the Amazon infrastructure stack was originally built for hosting Amazon's services. By offering this infrastructure as a service, Amazon is able to exploit its infrastructure and offer a new service to its customers at low cost.

Drawbacks for IaaS cloud providers are:

- **Under- and overprovisioning**
  Cloud providers have to offer their resources as if they are unlimited to the cloud user. This means that a cloud provider needs to own enough resources in order to fulfill the needs of a cloud user. These needs, however, may vary every time. Underprovisioning of a data center causes that a cloud user might not be able to obtain the resources it asks for, since the cloud provider does not have enough machines

available. Overprovisioning is extremely expensive, since servers are bought and maintained, but are not used.

Platform-as-a-Service (PaaS)

Platform-as-a-Service is a service model in which users are offered a platform on which they can develop and deploy their applications. The platform offers support for using resources from the underlying infrastructure. Platforms are mostly built for a certain domain such as, e.g., development of web applications, and are programming language-dependent.

There are several cloud platforms available nowadays. Microsoft offers the Windows Azure platform, which can be used for developing (web) applications and services based on the .NET framework. Google's App Engine is a platform for the development and deployment of Go, Python and Java-based (web) applications.

Benefits of PaaS for cloud users are:

- **Development platform**
  PaaS offers cloud users a platform on which they can manage and deploy their applications. Instead of having to manage issues such as scalability, load balancing and data management, cloud users can concentrate on application logic.
- **No hardware and server management needed**
  Customers can deploy applications relatively easily on the platform, since no network administrators are necessary for installing and maintaining servers or virtual machines.

Drawbacks of PaaS for cloud users are:

- **Forced to solutions in the cloud**
  PaaS offers combinations of services. For example, Windows Azure provides users with a .NET environment. The platform offers support for databases in the form of SQL Azure. Application developers can choose to use a different database, but have to perform difficult operations to install these services on the platform, or have to host the database by a third-party. PaaS users are more or less forced to use the solutions that are offered by the cloud provider in order to get the full benefits from the platform.

Benefits for PaaS cloud providers are:

- **Focus on infrastructure and platform**
  The software that runs on the platform is managed by the cloud user and the cloud

provider is responsible for the infrastructure and the platform.

Drawbacks for PaaS cloud providers are:

- **Platform development**
  The platform that is offered by the cloud provider is a piece of software. Complex software is needed to offer services such as automatic scalability and data replication. Faults in the platform can lead to failure of customer applications, so the platform has to be fault tolerant and stable.

Software-as-a-Service (SaaS)

With Software-as-a-Service, cloud providers offer an application that is deployed on a cloud platform. Users of the application access the application through the Internet, often using a browser. One of the benefits of SaaS is that cloud providers are able to manage their software from inside their company. Software is not installed on the computers of the cloud users, but instead runs on the servers of the cloud provider. When a fault is detected in the software, this can be easily fixed by repairing the software on the server, instead of having to distribute an update to all the users.

There are several examples of Software-as-a-Service. For example, Google offers several web applications, such as Gmail and Google Docs, as online services. Another example is SalesForce.com, which offers CRM online solutions as a service.

Benefits of SaaS for cloud users are:

- **Pay-per-use**
  Instead of having to purchase a license for each user of an application, organizations are billed based on the usage of the software. A couple of years ago software was often purchased on a pay-per-license base. Network administrators had to install applications on the workstations of a cloud user's company and for each application instance the cloud user had to pay for a license, even if the user of a particular workstation did not use the application. With pay-per-use, cloud users pay only for the users and the usage time of the application.

- **Updates**
  Applications in the cloud are managed by a cloud provider. The cloud provider is able to perform updates to the software directly in the cloud. Instead of having to distribute updates to the cloud user, the users always work with the most actual version since they access the application in the cloud.

Drawbacks of SaaS for cloud users are:

- **Data lock-in**
  Data lock-in is one of the typical problems of SaaS. In case cloud users decide to work with another application of another provider, it might be hard to move the data to this other application. Not every application provider stores data in a standardized way and interfaces for retrieving all the data from an application may not be available.

Benefits for SaaS cloud providers are:

- **Maintenance**
  Maintenance can be directly performed in the cloud application itself. Updates do not have to be distributed to the cloud users but are directly applied upon the software in the cloud.

Drawbacks for SaaS cloud providers are:

- **Infrastructure needed**
  In traditional software deployment, software is shipped to the user. The hardware on which the application is installed is managed by the user. With cloud computing, the software runs on servers of the cloud provider. This means that cloud providers have to perform infrastructure maintenance, or they have to rent infrastructure or a platform for hosting their applications.
- **Responsibility**
  Applications that run in the cloud are managed by the SaaS provider. When the application in the cloud is not accessible or not working any more because of erroneous updates or changes in the software, cloud users are not able to work with the software any more. It is a big responsibility for cloud providers to make sure the software is kept up and running.

2.2.3 Cloud types

The cloud types identified in [10][3] are discussed below.

Public Cloud

A public cloud is provisioned for exclusive use by the general public. Cloud users access the cloud through the Internet. Public clouds are widely available nowadays. For example, companies as Microsoft, Google and Amazon offer public cloud computing services. The biggest benefit of public clouds is that the management of the servers is provided by the

third-party provider. Users just pay for the usage of the cloud, and issues as scalability and replication are handled by the cloud provider.

Private Cloud

Private clouds are for exclusive use of a single organization. Private clouds can be hosted inside or outside the cloud user's organization and managed by the cloud user's organization itself or by a third-party provider. This form of cloud computing can be used when cloud users have to deal with strict security concerns, in case data has to be hosted inside the cloud user's organization.

Hybrid Cloud

Hybrid clouds are created by combining a private and a public cloud. With hybrid clouds, organizations can choose to store their critical data inside the company using a private cloud, while the less critical data and services can be stored in the public cloud. The hybrid cloud approach benefits from the advantages of both public and private clouds. Scalability is maintained, since the public cloud is used for offering the services, while data security is maintained by storing critical data in the private cloud.

Community Cloud

A community cloud is available for a specific community. Several companies that deal with the same concerns may decide to host their services together, in order to collaborate. Community clouds can be managed by one or more organizations within the community, but the cloud may alternatively be hosted by a third-party provider.

2.3 BPM in the cloud

Cloud-based BPM gives cloud users the opportunity to use cloud software in a pay-per-use manner, instead of having to make upfront investments on BPM software, hardware and maintenance [4]. Systems scale up and down according to the cloud users needs, which means that the user does not have to worry about over-provisioning or under-provisioning.

Privacy protection is one of the barriers for performing BPM in the cloud environment. Not all users want to put their sensitive data in the cloud. Another issue is efficiency. Computation-intensive activities can benefit from the cloud because of the scalability of the cloud. Activities that are not computation-intensive, however, do not always benefit

from cloud computing. The performance of an activity that is running on-premise might be higher than in the cloud because of data that needs to be transferred to the cloud first in order to perform the activity. These activities can also make cloud computing expensive, since data transfer is one of the billing factors of cloud computing.

2.3.1 Combining traditional and cloud-based BPM

In most BPM solutions nowadays, the process engine, the activities and process data are placed on the same side, either on-premise or the cloud. The authors of [4] investigated the distribution possibilities of BPM in the cloud by introducing a PAD model, in which the process engine, the activities involved in a process and the data involved in a process are separately distributed, as shown in Figure 2.4. In this figure, P stands for the process enactment engine, which is responsible for activating and monitoring all the activities, A stands for activities that need to be performed in a business process, and D stands for the storage of data that is involved in the business process. By making the distinction between the process engine, the activities and the data, cloud users gain the flexibility to place activities that are not computation-intensive and sensitive data at the user-end side and all the other activities and non-sensitive data in the cloud.

The PAD model introduced in [4] defines four possible distribution patterns. The first pattern is the traditional BPM solution where everything is placed at the user-end. The second pattern is useful when a user already has a BPM system on the user-end, but the computation-intensive activities are placed in the cloud to increase their performance. The third pattern is useful for users who do not have a BPM system yet, so that a cloud-based BPM system can be utilized in a pay-per-use manner and activities that are not computation-intensive and sensitive data can be placed at the user-end. The fourth pattern is the cloud-based BPM pattern in which all elements are placed in the cloud.

Business processes define two types of flows, namely a control-flow and a data-flow. The control-flow regulates the activities that are performed and the sequence of these activities, while the data-flow determines how data is transferred from one activity to another. BPM workflow engines have to deal with both control-flows and data-flows. A data-flow might involve sensitive data, therefore, when a BPM workflow engine is deployed on the cloud, data-flows should be protected.

In the architecture proposed in [4], the cloud side engine only deals with data-flow by using reference IDs instead of the actual data. When an activity needs sensitive data, the transfer of the data to the activity is handled under user surveillance through an encrypted tunnel. Sensitive data is stored at the user-end and non-sensitive data is stored in the cloud. An overview of the architecture proposed in [4] is shown in Figure 2.5.

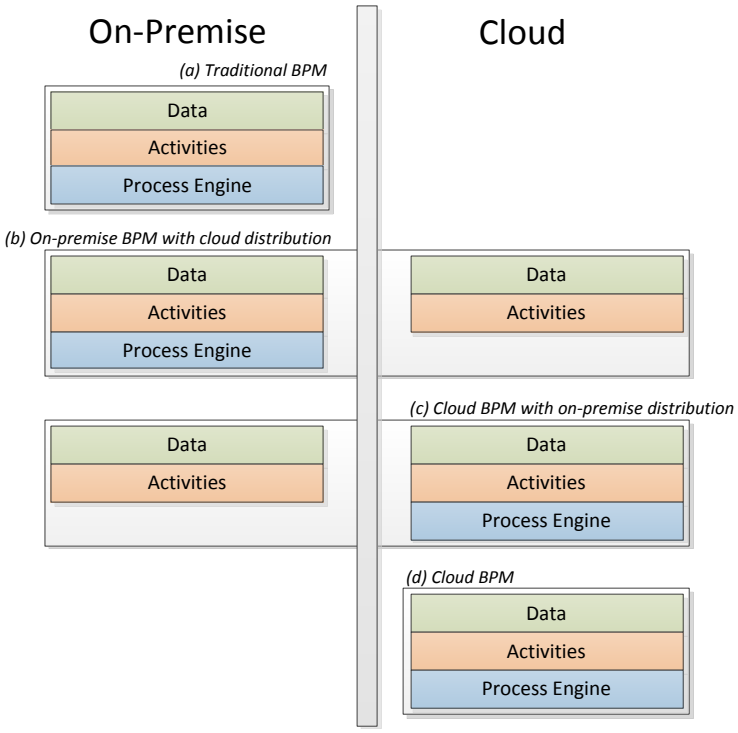


Figure 2.4: Patterns for BPM placement, based on [4]

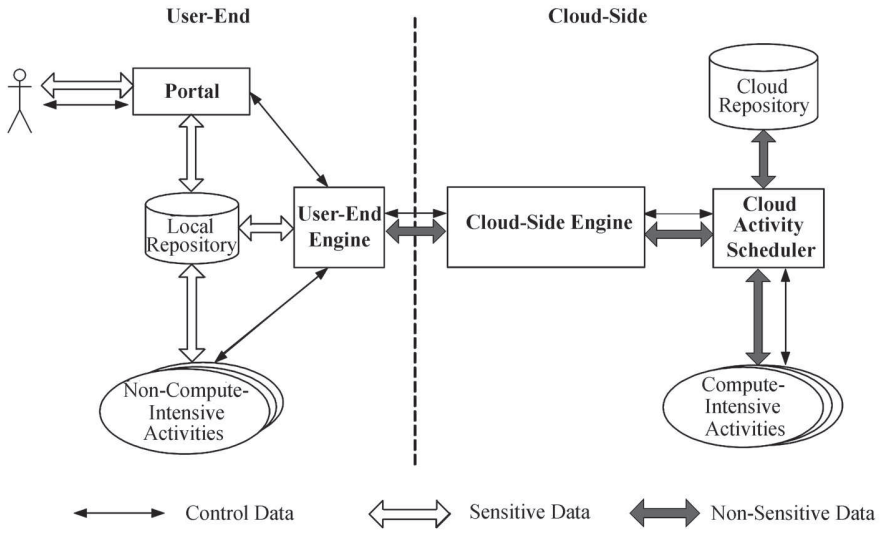


Figure 2.5: Architecture of a cloud-based BPM system combined with user-end distribution [4]

The costs for using cloud computing are investigated in several articles [2, 11]. In [4], formulas are given for calculating the optimal distribution of activities, when activities can be placed in the cloud or on-premise. The calculation takes into account the time costs, monetary costs and privacy risk costs. By using these formulas, cloud users can

make an estimation of the costs of deploying parts of their application on-premise and in the cloud.

3. Approach

In this chapter we introduce the approach we have taken in this research project. This chapter is structured as follows. Section 3.1 explains the general development goals. Section 3.2 discusses related work on decomposition of business processes. Section 3.3 introduces the transformation chain we use for the transformations.

3.1 General development goals

In this research we extend the work of [4] by focusing on the decomposition of business processes into collaborating processes for distribution on-premise or in the cloud.

We identify a fifth pattern for a PAD model, in which process engines, activities and data are placed in both the cloud and on-premise. This extension of the model is shown in Figure 3.1.

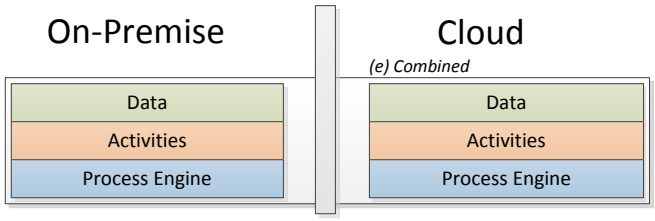
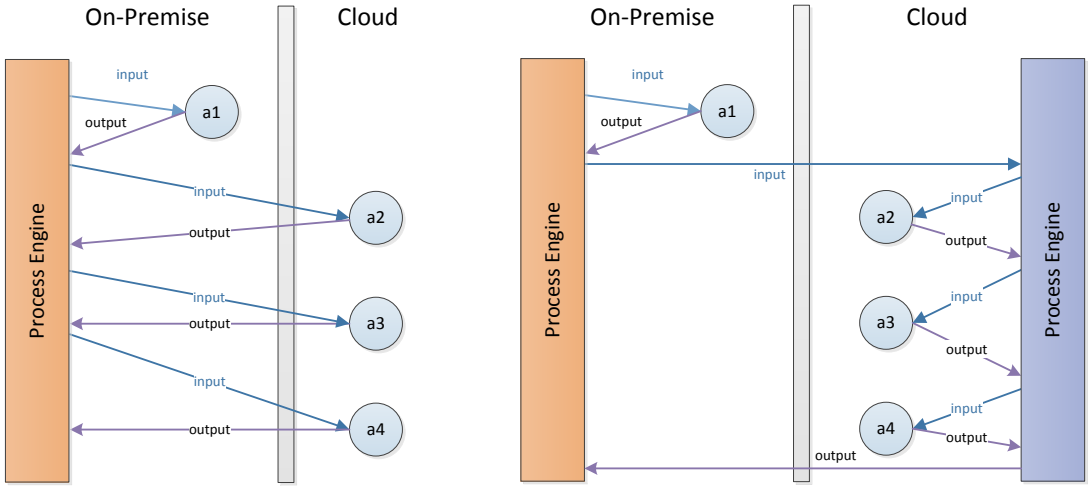


Figure 3.1: Fifth PAD pattern

The architecture proposed in [4] also considers process engines on both the cloud and on-premise sides, but the decomposition of the original process is not addressed there. In our approach, we want to make use of two separate process engines to minimize the amount of data that has to be exchanged between the cloud and on-premise. Each process engine regulates both the control-flows and data-flows of a process.

Consider a process engine in which the output of one activity is the input for the following activity. Figure 3.2a shows a situation in which a process is executed by a single process engine situated on-premise, where some of the activities within the process are placed in the cloud. Since the process is coordinated by the process engine, data is not directly sent from activity to activity, but instead is sent to the process engine first. In case of adjacent cloud activities, using one process engine on-premise leads to unnecessary data exchange between the process engine and the cloud. By introducing a second process engine in the cloud, we can avoid this problem. Adjacent activities with the same distribution location do not have to send their data from cloud to on-premise, or vice versa, since the coordination can be performed by the process engine in the cloud. This situation is shown in Figure 3.2b.



(a) One process engine (b) Two process engines

Figure 3.2: Data transfer between activities coordinated by process engines

Our overall goal is to create a transformation framework in which users can automatically decompose a business process into collaborating business processes for distribution on-premise and in the cloud, based upon a list in which activities and data are marked with their desired distribution location. In addition, users should be able to define data restrictions, to ensure that sensitive data stays within the premises of an organization. A schematic overview of the transformation is shown in Figure 3.3.

3.2 Related Work

The purpose of this section is to identify techniques that can be applied for the decomposition of business processes. Below, we discuss related work on the decomposition of orchestrations. Several research groups have investigated the possibility of decentralizing orchestrations. In a centralized orchestration, a process is coordinated by a single orchestrator. Decentralized orchestrations are distributed among several orchestrators. By distributing parts of a process over separate orchestrators, the message overhead may be reduced, which potentially leads to better response time and throughput [12].

In [13, 12, 14, 15, 16], new orchestrations are created for each service that is used within the business process, hereby creating direct communication between services, instead of being coordinated by one single orchestrator. The business processes are defined in BPEL [6]. Not only decomposition is defined, but also analysis on synchronization issues is performed. The work captures a BPEL process first in a control-flow graph, which is used in turn to create a Program Dependency Graph (PDG) [17]. The transformations are per-

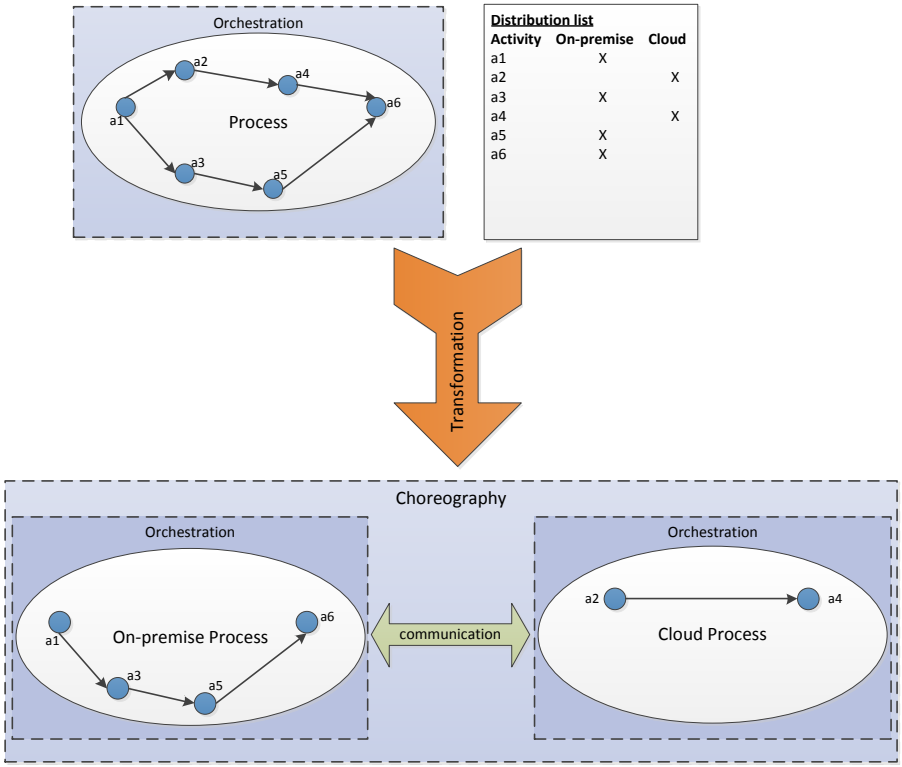


Figure 3.3: Example of decomposition

formed on PDGs and the newly created graphs are transformed back into BPEL. The partitioning approach is based on the observation that each service in the process corresponds to a fixed node and for each fixed node a partition is generated. In our approach we want to create processes in which multiple services can be used. This partitioning algorithm is therefore not suitable to our approach.

Research in [18, 19, 20] focuses on decentralization of orchestrations by using BPEL processes. The main focus of the research is to use Dead Path Elimination (DPE) [6], for ensuring the execution completion of decentralized processes. Using DPE also leads to very specific language-related problems, therefore these research papers are only useful when BPEL is selected as the input and output language of our transformation framework.

In [21], decentralization of BPEL processes is considered. The authors use a graph transformation approach for transforming the BPEL process. The transformation rules are not defined in the paper. The type graph with which the graph transformations are performed is described and might be applicable to our situation.

In [22], the authors state that the current research on decentralizing orchestrations focuses too much on specific business process languages. In most cases, implementation level languages, such as BPEL [6], are used. In our situation, the decision for distributing activities

and data to the cloud is not only based on performance issues, but also on safety measures, regulated by the organization or government. The decision to execute an activity on-premise or in the cloud might therefore be already taken in the design phase of the BPM lifecycle.

3.3 Transformation chain

Instead of building a solution for a specific business process language, we opted for using an intermediate model in which the structure and semantics of business processes are captured. There are two reasons for using an intermediate model:

1. A business process is defined in a business process language using the syntax of the language. The decomposition transformations we want to apply should comply to the semantics of the business process language. We therefore need to lift the original business process to a model in which the intended semantics of the model are preserved.

2. By using an intermediate model, we can purely focus on the decomposition problems, without having to consider language specific problems. As a drawback, extra transformations are needed for converting a business process to the intermediate model and back.

Our approach consists of a transformation chain with 3 phases: a lifting phase, a decomposition phase and a grounding phase. An overview of our approach is shown in Figure 3.4.

Transformation 1: Lifting
The lifting transformation transforms a business process defined in some business process language into an instance of the intermediate model. Data analysis is performed during this transformation phase to capture data dependencies between activities in the process. This information is needed for ensuring that no data restrictions are violated during the decomposition transformation.

Transformation 2: Decomposition
The decomposition transformation transforms an instance of the intermediate model according to an activity distribution list into a new instance of the intermediate model that represents the decomposed processes and the communication between the processes. The activity distribution list defines the distribution locations of each of the activities in the resulting process. Furthermore, data restrictions can be defined in the list. The distribution location of each data element used within the

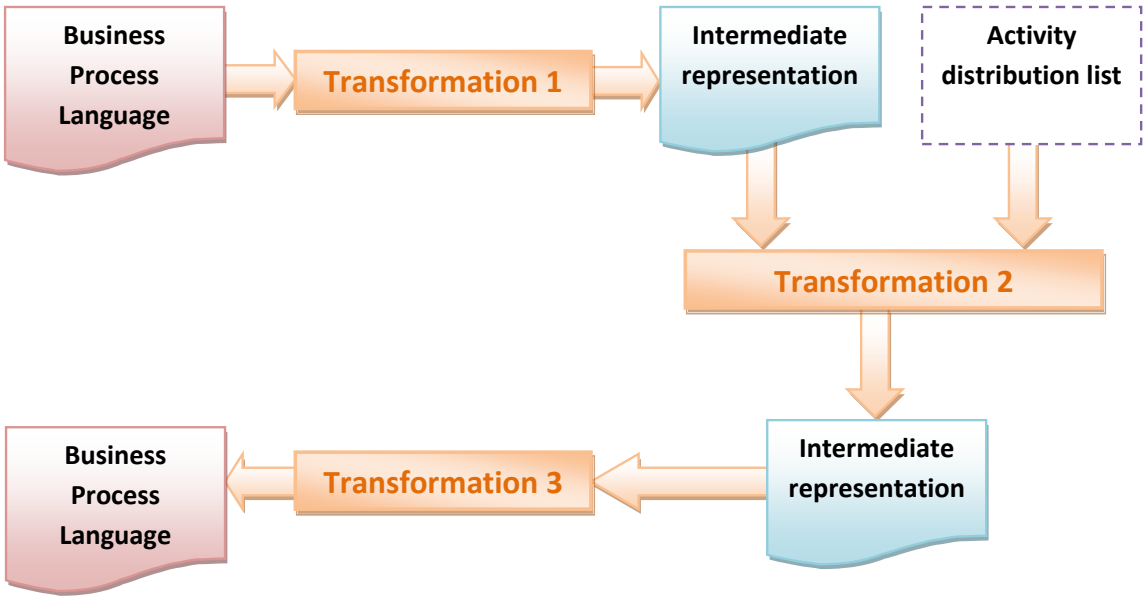


Figure 3.4: Schematic representation of the transformation chain

process can also be defined, to ensure that the data element stays within the borders of the defined location.

Transformation 3: Grounding

The grounding transformation transforms a decomposed intermediate model back to an existing process language. Depending on the language which is used, the transformation creates separate orchestrations for each of the processes and optionally a choreography in which the cooperation between the processes is described.

4. Intermediate Model

This chapter defines the intermediate model we have used for the decomposition transformation. Section 4.1 defines the requirements that should be fulfilled by the intermediate model. Section 4.2 compares several existing models to identify a suitable representation. Section 4.3 defines our intermediate model by using graphical examples. Section 4.4 formally defines our intermediate model. Section 4.5 shows how concepts from WS-BPEL [6] are mapped to the intermediate model.

4.1 Requirements

The challenge for our intermediate model is to use a model which is reasonably simple, but is still able to capture complex business process situations.

We used the control-flow workflow patterns defined in [23] for selecting the most common workflow patterns. We decided not to support all of the control-flow workflow patterns at first, since the intermediate model would get too complex. Instead, we identified the patterns that are present in the business process languages WS-BPEL [6], WS-CDL [7], Amber [24] and BPMN 2.0 [5, 25]. From the identified patterns, the most common patterns were selected and used as requirements for the intermediate model. In future work, the intermediate language can be extended to support more control-flow workflow patterns. The following patterns should at least be supported by our intermediate model:

WP1: Sequence
The intermediate model should have a mechanism for modeling control flows, in order to be able to express the sequence of execution of activities within a process.

WP2: Parallel Split
The intermediate model should support parallel execution of activities. A construct is needed for splitting up a process into two or more branches, which are executed simultaneously.

WP3: Synchronization
The intermediate model should have a mechanism for synchronizing two simultaneously executing branches. A synchronization construct is needed in which multiple branches are joined into one executing branch.

WP4: Conditional Choice
The intermediate model should have a construct for executing a branch, based upon an evaluated condition.

WP5: Simple Merge
The intermediate model should have a construct for joining multiple alternative branches, from which one is executed.

WP10: Arbitrary Cycles
The intermediate model should support a construct for modeling recursive behavior.

The requirements identified so far are all based on control-flows. In addition, the following requirements should also be supported by our intermediate model:

Data dependencies
Since we might have to deal with sensitive data, it is crucial that the consequences of moving activities around are measurable. By explicitly representing data dependencies between nodes, the flow of data through the process can be monitored.

Communication
Since the original process needs to be split up into collaborating processes, there should be a communication mechanism for describing that one process invokes another.

4.2 Model selection

We compared existing models for their suitability to support the requirements of our intermediate model. The models we compared were mainly taken from similar decentralization approaches. The following models were considered: Program Dependency Graphs (PDG) [17], Control Flow Graph [12], Protocol Tree [26] and Petri nets [27].

We analyzed all of the models and came to the following conclusions:

Program Dependency Graph
Program Dependency Graphs support data dependencies between nodes. Control dependencies however, are not directly visible in these graphs. This means that complex behaviors, such as parallel execution of nodes cannot be described by a PDG.

Control Flow Graph
Control Flow Graphs (CFG) can be used for modeling the control flow within a process. The data dependencies between nodes however, are not visible in these graphs.

(Colored) Petri nets
Traditional Petri nets are not able to support all requirements we set for our intermediate model. For example, data dependencies cannot be modeled in traditional Petri nets. Data dependencies can be modeled thought in Petri net variants such as Colored Petri Nets [28]. The downside of using Petri nets for modeling processes

is that many different nodes are needed for representing a process. A transition between two nodes is modeled with places, transitions, tokens and arrows, which would bring overhead to our intermediate model.

Protocol Tree

Protocol Trees are able to capture only block-structured processes. Since we also want to be able to capture graph-based structures, Protocol Trees are not directly suitable to support our requirements.

Since none of the selected models completely satisfies our defined requirements, we decided to define our own intermediate model. Our model is based upon Control Flow Graphs, Program Dependency Graphs and Protocol Trees. The structure of a Control Flow Graph is used for defining control-flows between nodes, and the model also contains data-dependency edges to capture data-flows. The formal definition of Protocol Trees is used as basis for the formal definition of our intermediate model.

4.3 Model definition

We use a graph-based representation for processes, since the base languages we targeted are either block-structured or graph-structured [29].

A graph consists of nodes and edges. In our model, a node represents either an activity or a control element. An edge defines a relation between two nodes. In order to be able to capture complex constructs and data dependencies between nodes, we introduce multiple specializations of nodes and edges. For each of the nodes and edges we also define a graphical representation.

4.3.1 Node types

Activities

Activities can be modeled by activity nodes. Every activity node has at most one incoming control edge and at most one outgoing control edge.

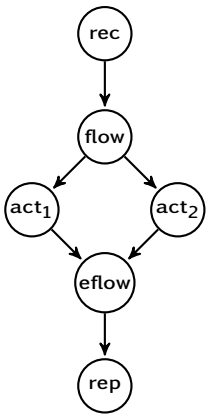
Parallel behavior

A process with parallel behavior can be modeled using flow and end-flow nodes. A flow node splits an execution branch into multiple branches, which are executed simultaneously. The minimum number of outgoing control edges of a flow node is two. Multiple

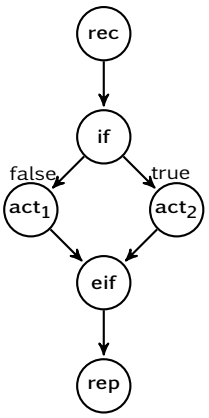
parallel branches can be joined into one execution branch by using the end-flow node. The end-flow node has two or more incoming control edges and at most one outgoing control edge. An example of parallel behavior modeled in the intermediate model is shown in Figure 4.1a.

Conditional behavior

Branch selection based upon an evaluated condition can be modeled by using conditional nodes. The conditional node (if-node) has two outgoing control edges. One edge is labeled with "true" and is used when the evaluated condition yields true. The other edge is labeled with "false" and is used otherwise. After the condition in the if-node is evaluated, only one of the outgoing branches can be taken. Conditional branches can be joined by using an end-conditional node (eif-node). This node converts multiple incoming branches into one outgoing branch. An example of conditional behavior modeled in the intermediate model is shown in Figure 4.1b.



(a) Parallel behaviour



(b) Conditional behaviour

Figure 4.1: Modeling parallel and conditional behaviors

Loops

We defined one single node for modeling repetitive behavior in the intermediate model, the so called loop-node. A loop-node evaluates a loop-condition and according to the result of the evaluation, the loop branch is either executed or denied. The loop-node is comparable to the if-node, since it also evaluates a condition and has outgoing "true" and "false" edges. The outgoing branches, however are never joined. Instead, one of the branches ends with an outgoing edge back to the loop-node. This branch is called the loop-branch. The other branch points to the behavior which should be executed as soon as

the loop-condition does not hold anymore.

The loop-node can be placed in the beginning or at the end of the loop-branch. The first situation results in zero or more executions of the loop-branch, since the loop-condition needs to be evaluated before the loop-branch is executed. In the second situation, the loop-branch is executed at least once, since the loop-condition is evaluated after execution of the loop-branch. An example of both scenarios is shown in Figure 4.2.

(a) Loop-node before loop-branch (b) Loop-node in the end of the loop-branch

Figure 4.2: Modelling loops

Communication

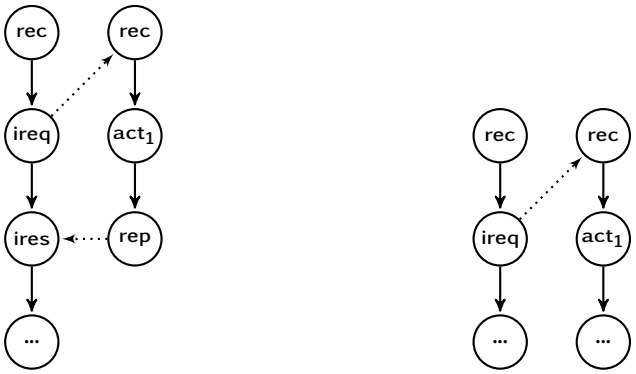
Communication nodes model communication between two processes. The intermediate model supports four possible communication nodes: invoke-request, invoke-response, receive and reply. These nodes can be used to model synchronous and asynchronous communication.

The invoke-request-node (ireq-node) is used for invoking a process. The node has one outgoing communication edge, which points to a receive-node, located in the process that is invoked. The invoke-request-node does not wait until the execution of the invoked process is finished, instead it proceeds to the successor node.

The invoke-response-node (ires-node) is used as a synchronization node for communication with other processes. The node has one incoming communication edge, which originates from a reply-node located in another process. The invoke-response-node waits for the response from the other process, before continuing its execution.

In case of synchronous communication, a process (P1) uses an invoke-request-node to invoke another process (P2). The invoke-request-node follows its outgoing control edge, which is connected to an invoke-response-node. This node, in turn, waits until process P2

is finished, before continuing with process P1. In asynchronous communications, a process (P1) invokes another process (P2) by using an invoke-request-node. After calling P2, execution of P1 continues. Both situations are shown in Figure 4.3.



(a) Synchronous communication (b) Asynchronous communication

Figure 4.3: Synchronous and asynchronous communication in the intermediate representation

4.3.2 Edge types

Our intermediate model distinguishes between control edges, data edges and communication edges.

Control edges

Control flow is modeled in the intermediate model by control flow edges, which are represented by solid arrows in our graphical notation. The node from which the edge originates triggers the edge as soon as the execution of the nodes action has been finished. The node in which the edge terminates waits for a trigger, caused by an incoming edge, before it starts executing the action of the node. A control edge can be labeled with "true" or "false", in case the control edge originates from a conditional-node. When the evaluated condition matches the label of the edge, the edge is triggered by the conditional-node.

Data edges

In Business Process Languages such as WS-BPEL [6], data flow between activities is defined implicitly. Instead of sending data from activity to activity, activities can access variables directly, provided that the activity has access to the scope in which the variable is defined. By introducing data edges in our intermediate model, we are able to investigate

the consequences for data exchange of moving activities from one process to another. This information is needed to verify if no data constraints are violated during the transformation. A data link is represented by a dashed arrow between two nodes in our graphical notation. A data edge from node N1 to node N2 implies that data defined in node N1 is used in node N2. Each data edge is provided with a label, in which the name of the shared data item is defined.

Communication edges

Communication edges are defined between communication nodes. A communication edge sends control and data to a different process. Communication edges are labeled with the names of the data items that are sent over the edge.

4.4 Formal definition

In this section we define our intermediate model formally. We refer to concepts introduced in Section 4.3. Formally, our intermediate model I is a tuple with the following elements $(A, C, S, ctype, stype, E, L, nlabel, elabel)$ where:

- A is a set of activity nodes.
- C is a set of communication nodes.
- S is a set of structural nodes (flow nodes, end-flow nodes, if nodes, end-if nodes and loop nodes).
- $ctype : C \rightarrow \{InvokeRequest, InvokeResponse, Receive, Reply\}$ is a function that assigns the communicator type to a communication node.
- $stype : S \rightarrow \{Flow, EndFlow, Conditional, EndConditional, Loop\}$ is a function that assigns a control node type to a control node.
- E is the set of all edges in the graph. Let $E = E_{ctrl} \cup E_{data} \cup E_{com}$. An edge is defined by a tuple $(n_1, etype, n_2)$ where $etype \in \{Control, Data, Communication\}$ is the type of the edge and $n_1, n_2 \in A \cup C \cup S$.
  - E_{ctrl} is the set of control flow edges. Let $e = (n_1, Control, n_2)$ where $n_1, n_2 \in \{A \cup C \cup S\}$ and $e \in E_{ctrl}$.
  - E_{data} is the set of data edges. Let $e = (n_1, Data, n_2)$ where $n_1, n_2 \in \{A \cup C \cup S\}$ and $e \in E_{data}$.

- E_{com} is the set of communication edges. Let $e = (n_1, Communication, n_2)$ where $n_1, n_2 \in C$ and $e \in E_{com}$.

- L is the set of text labels that can be assigned to nodes and edges.
- $nlabel : N \rightarrow L$, where $N = A \cup C \cup S$ is a function which assigns a textual label to a node.
- $elabel : E \rightarrow L$ is a function which assigns a textual label to an edge.

4.5 Mapping example

In this section we show two examples of constructs in WS-BPEL [6] and how they are mapped to the intermediate model.

The first example shows how a sequence construct in WS-BPEL is mapped to the intermediate model. The BPEL fragment is shown in Listing 4.1. A graphical representation of the intermediate model obtained after the mapping is shown in Figure 4.4.

```
<sequence>
  <receive ... variable="A" />
  <assign>
    <copy>
      <from>bpel:doXslTransform(..., $A)</from>
      <to variable="B" />
    </copy>
  </assign>
  <reply ... variable="B" />
</sequence>
```

Listing 4.1: BPEL sequence example

The receive and reply elements in the BPEL fragment are mapped to communication nodes and the assign element in BPEL is mapped to an activity node. In the BPEL example, variable A is received and a part of the variable is copied to variable B by the assign element. Eventually, variable B will be returned by the reply element. The data dependency edges in the intermediate model show the data dependencies between the nodes.

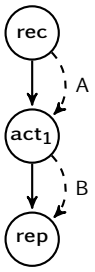


Figure 4.4: Graphical representation of the intermediate model generated from the first BPEL example.

The second example shows how a loop and a flow construct in WS-BPEL are mapped to the intermediate model. The BPEL specification is shown in Listing 4.2. A graphical representation of the obtained intermediate model is shown in Figure 4.5.

```
<sequence>
  <receive ... variable="A" />
  <while>
    <condition>...</condition>
    <flow>
      <invoke name="act1" inputVariable="A" ... />
      <invoke name="act2" inputVariable="A" outputVariable="B" />
    </flow>
  </while>
  <reply ... />
</sequence>
```

Listing 4.2: BPEL loop example

The while element in the BPEL example is mapped towards a loop construct in which the condition is evaluated before execution of the loop branch. The loop branch consists of a flow element, which is mapped in the intermediate model to a parallel construct with a flow and an end-flow node. The invokes that are executed within the parallel construct are mapped to communication nodes. The invocation with name "act1" is mapped to an asynchronous invocation element, since it expects no response. The invocation with name "act2" is mapped to two synchronous invocation nodes, since the invocation needs to wait for a response from the invoked service. Data dependencies are introduced between the receive node and the invocation nodes, since the invocation nodes use the variable that was received by the receive element.

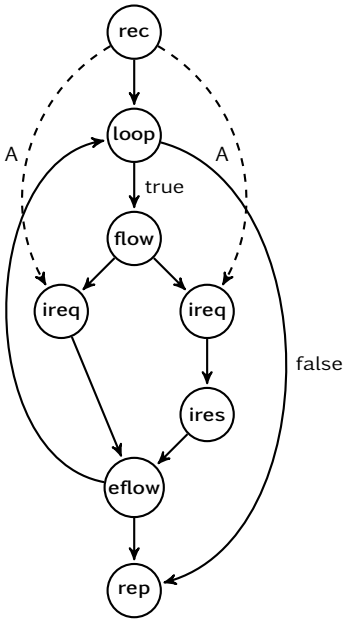


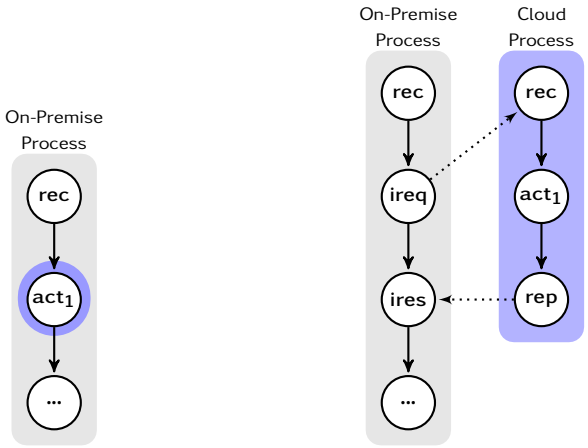
Figure 4.5: Graphical representation of the intermediate model generated from the second BPEL example.

5. Decomposition Analysis

The goal of this chapter is to identify possible transformations for each of the constructs defined in the intermediate model. In this analysis we take into account processes that are hosted on-premise and have activities that should be allocated in the cloud, or vice-versa.

5.1 Single activity

When a single activity is marked for allocation to the cloud (shown in Figure 5.1a), the solution shown in Figure 5.1b is suitable. In the solution, the activity is moved to a new cloud process and called in the on-premise process by synchronous invocation nodes. By using synchronous invocation, the execution sequence of the processes can be maintained, since the node following the activity in the original process has to wait until the cloud process is finished.



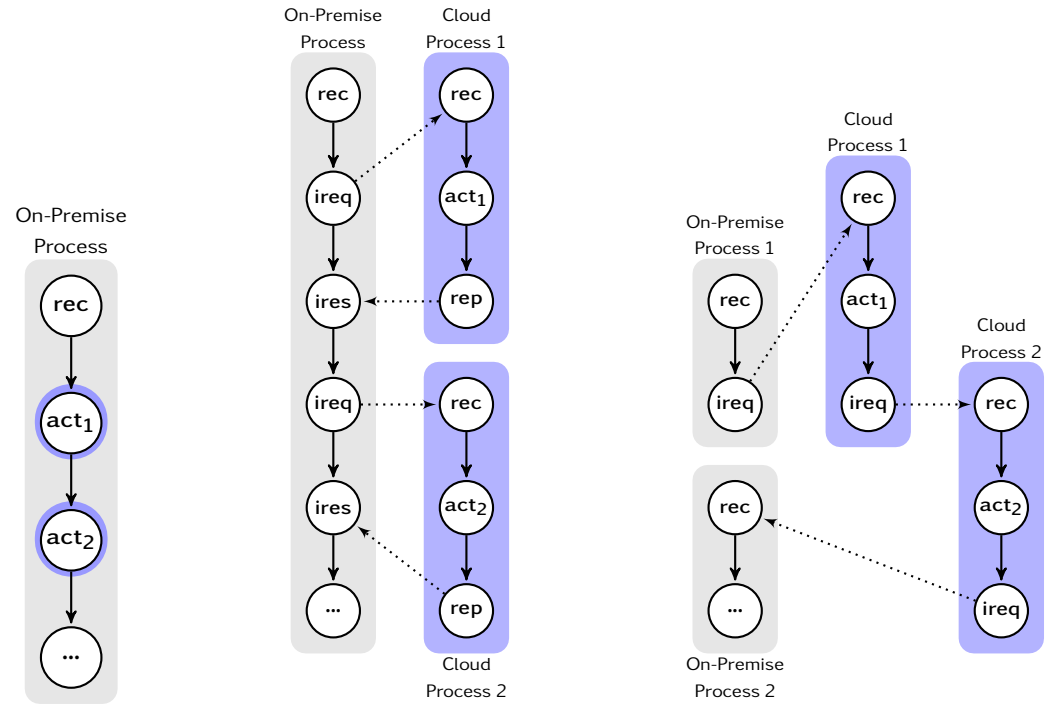
(a) situation (b) after

Figure 5.1: Moving a single activity from on-premise to the cloud

5.2 Sequential activities

When multiple sequential activities are marked for allocation to the cloud, the sequential activities can be placed in two separate cloud processes, or the sequential activities can be placed together in one cloud process. In this section we investigate four possible situations that are applicable when dealing with sequential activities.

We first discuss the allocation of sequential activities to separate processes, as shown in Figure 5.2. There are two possible solutions, which depend on the distribution of the



(a) situation (b) solution 1 (c) solution 2

Figure 5.2: Applying rule for single nodes to sequential nodes

control links between the activities:

- **Solution 1: Maintain control links on-premise**
  In the solution shown in Figure 5.2b, for each marked activity a new cloud process is created. Synchronous invocation nodes are introduced in the on-premise process for invoking the activities in the cloud.

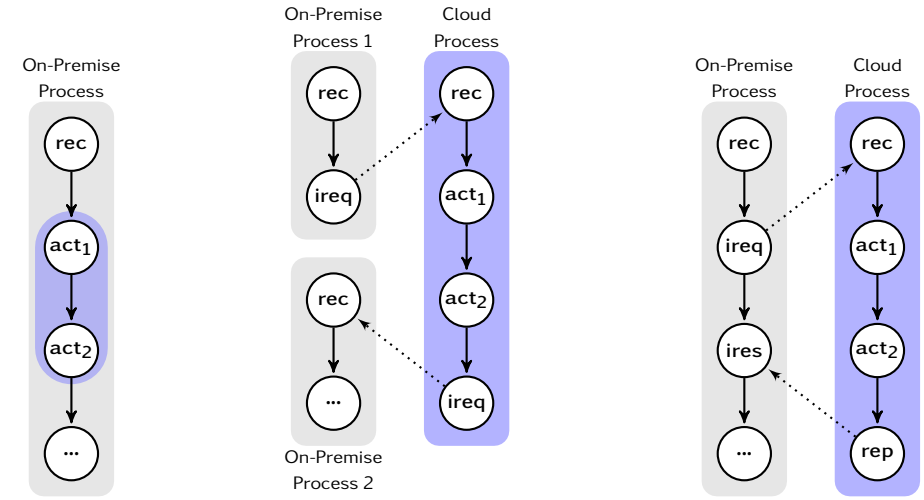
  In the original process, shown in Figure 5.2a, a control link is present between the activities. Since both activities are placed in new processes, there is no direct control link any more between the activities. In our first solution, a control link is introduced between the created communication nodes in the on-premise process, to keep the on-premise process together. The drawback of this solution is that there is unnecessary communication between the cloud and on-premise, since the result of the first cloud process is sent to the second cloud process via the on-premise process, instead of sending it directly.

- **Solution 2: Move control links to the cloud**
  In the second solution, shown in Figure 5.2c, both activities are moved to individual cloud processes. The on-premise process calls the first cloud process. After execution of the activity in the first cloud process, the second cloud process is called directly.

The second cloud process eventually gives a call back to the on-premise process.

The control link between the two activities in the original process, shown in Figure 5.2a, is moved to the cloud and placed between the invoke and receive of the first and second cloud process. As a consequence, the on-premise process is no longer a single process, but is decomposed into two separate processes.

Figure 5.3 shows the solutions for the second situation in which two sequential activities are moved together to one cloud process. The following two solutions are applicable in this situation:



(a) situation (b) solution 3 (c) solution 4

Figure 5.3: Moving sequential activities as a block

- **Solution 3: Splitting up on-premise processes**
  The first solution is to move the sequential activities to a new cloud process. By moving these activities, a gap arises between the nodes that are placed before the cloud process and the nodes after the cloud process. This solution is shown in Figure 5.3b.

  This solution leads to many processes, since every time a sequence of nodes is placed in the cloud, the on-premise process is split up.

- **Solution 4: Replace by synchronous invocation node**
  The second solution shown in Figure 5.3c replaces the moved part in the on-premise process with a control edge, hereby maintaining the structure of the on-premise process.

  Replication of the control link between the processes leads to more complex processes, but the overall structure of the on-premise process is maintained, since the

cloud nodes are replaced by invocation nodes. This makes the on-premise process more robust, since execution of the overall process is coordinated by the on-premise process.

5.3 Composite constructs

Parallel constructs and conditional constructs can be generalized as composite constructs. When looking at these constructs from a semantics perspective, their behavior is completely different. The syntax structure of both constructs is, however, quite similar. Both constructs start with a node which splits the process into several branches. Eventually, the branches join in an end-node, which closes the composite construct.

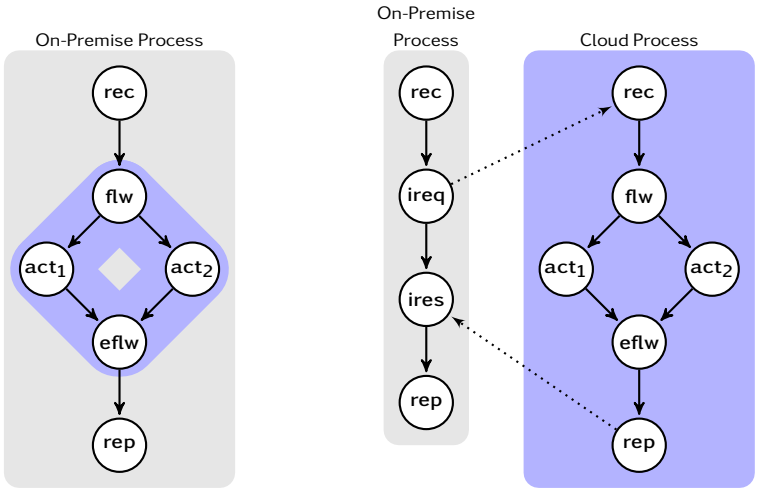
In this section we analyze all the decomposition possibilities for composite constructs. The possibilities are categorized in three categories:

1. The start and end node (e.g., flw and eflw) and all the contents within the composite construct are allocated to the same destination, and are kept together as a whole.
2. The start and end node have the same distribution location, but activities within the composite construct need to be maintained locally.
3. The start and the end node have different distribution locations.

Section 5.2 shows that when sequential activities need to be distributed in the cloud, the on-premise process can be either split up into individual processes or kept together. We acknowledge that this situation is not only applicable to sequential nodes, but also to composite nodes. In the remainder of this chapter we keep the on-premise process together, when the start and end node of a composite node have the same distribution location, to reduce the number of solutions. For activities within branches of the composite constructs, we move activities as a block and keep the surrounding process together, to reduce the number of solutions. The possible decomposition rules can be applied recursively on each of the branches of the composite constructs.

5.3.1 Category 1: Moving the composite construct as a whole

Figure 5.4a shows the situation where the start and end node and all the nodes in the branches of the composite node are marked for allocation to the cloud. Figure 5.4b shows the solution that is applicable in this situation. The construct is moved as a whole to a new cloud process. In the on-premise process, synchronous invocation nodes are introduced to call the cloud process.



(a) situation (b) solution

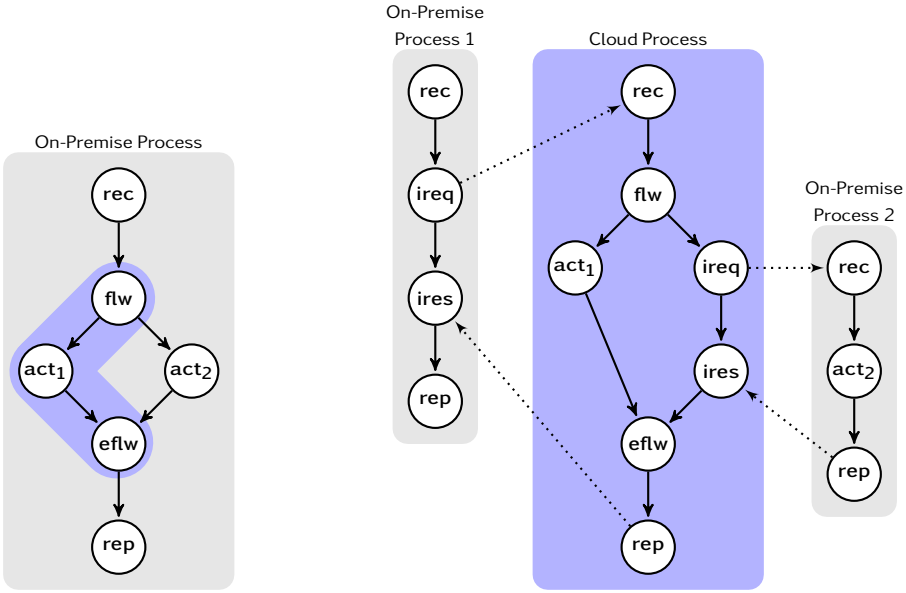
Figure 5.4: Moving the whole composite construct

5.3.2 Category 2: Start/end nodes with the same distribution location

Three possible situations exist with composite nodes, where the start and end node have the same distribution location, and contents within the construct have a different distribution location. For those situations we present a solution in which the composite construct itself is placed in only one process, either on-premise or in the cloud. The branches within the composite construct are treated as sub processes and the earlier defined rules are recursively applied on these sub processes. Activities within the branches of the composite construct with the same distribution location as the construct itself are placed directly within the construct. Activities with a different distribution location are placed in new processes.

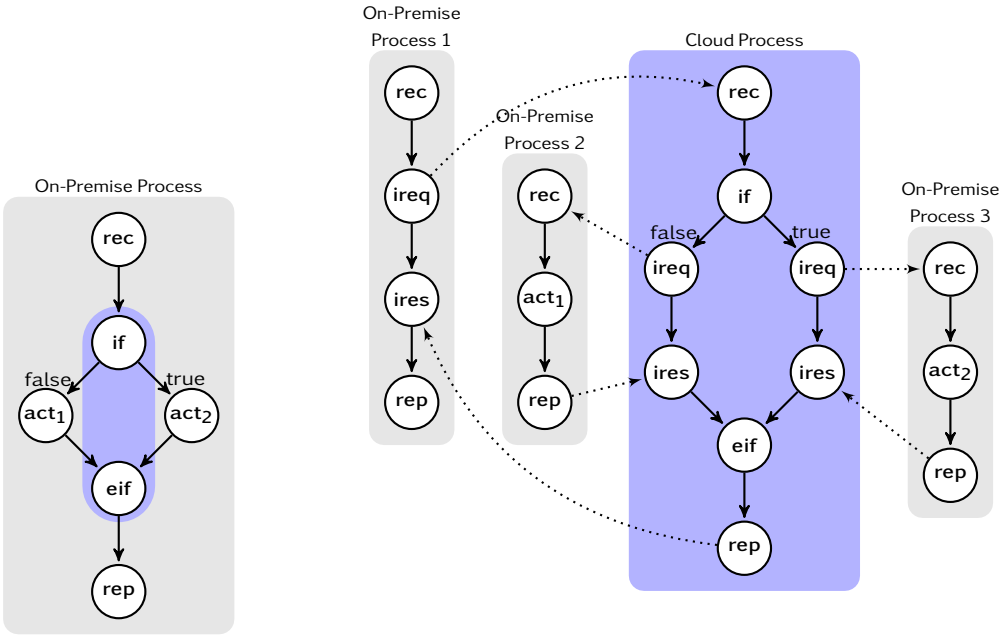
- **Composite construct marked, one branch stays on-premise**
  Figure 5.5a represents the situation where the composite nodes are marked for allocation to the cloud, but one branch should be executed on-premise. In the solution, a new cloud process is created in which the composite construct is placed. Invocation nodes are introduced in on-premise process 1 to invoke the cloud process. Activity 1 is placed directly within the construct. Activity 2 is placed in a new on-premise process and is called by invocation nodes in the cloud process.

- **Composite construct marked, all branches stay on-premise**
  Figure 5.6a shows the situation where the composite nodes are marked for distribution in the cloud, but the activities within the branches of the composite node need to be distributed on-premise. As a solution, we distribute the composite nodes and create subprocesses for each of the branches, as shown in Figure 5.6b.



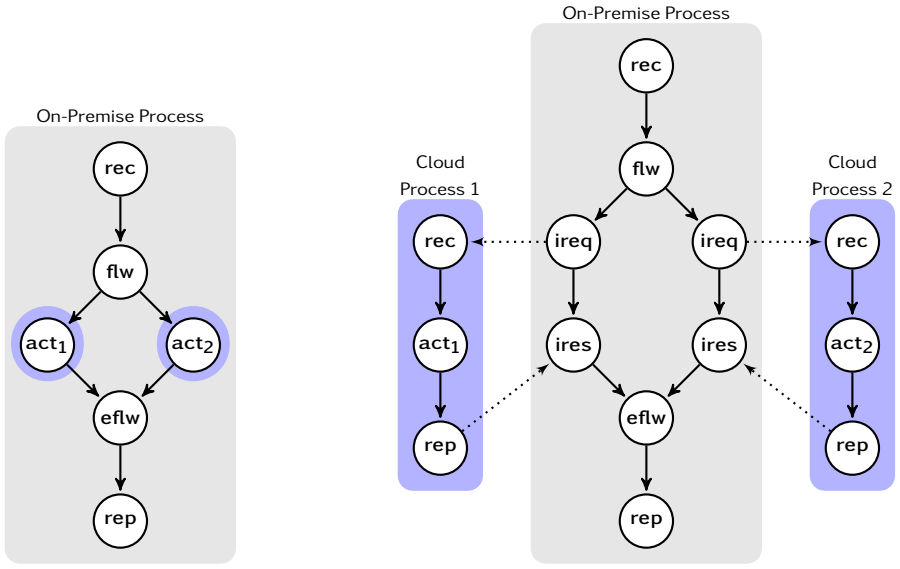
(a) situation (b) solution

Figure 5.5: Composite construct marked, one branch stays on-premise



(a) situation (b) solution

Figure 5.6: Composite construct marked, all branches stay on-premise



(a) situation (b) solution

Figure 5.7: Composite construct stays on-premise, activities in the branches marked for movement

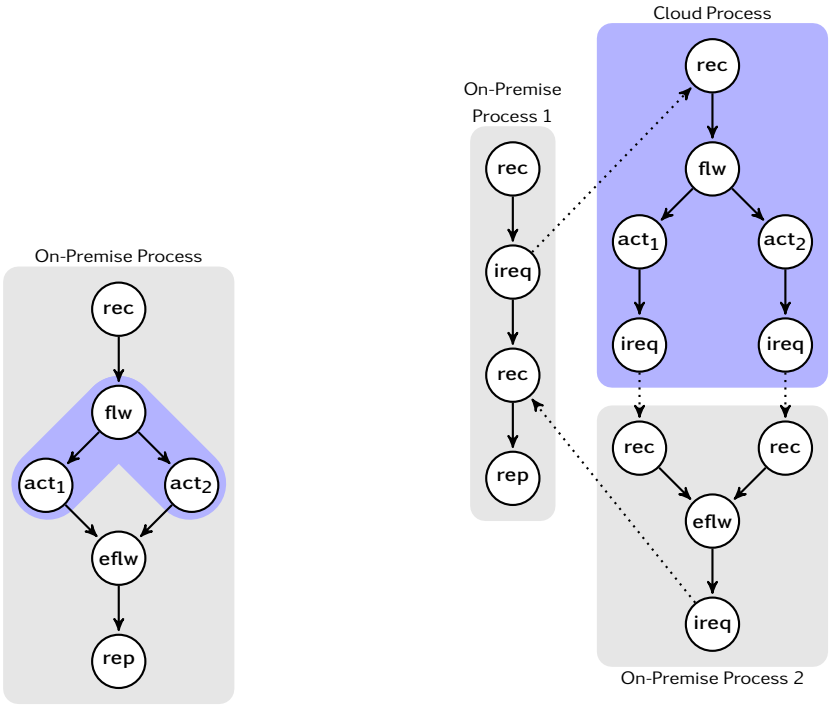
- **Composite construct stays on-premise, activities in the branches marked for movement**
  In the final situation, the composite construct is allocated on-premise, but the branches within the construct are marked for distribution in the cloud. This is the opposite of the previous situation. Invocation nodes are introduced in the branches and both activities are placed in separate cloud processes. The situation and the solution are shown in Figure 5.7.

5.3.3 Category 3: Start/end node with different distribution location

The last category consists of situations in which the start-node and the end-node of the composite construct have different distribution locations.

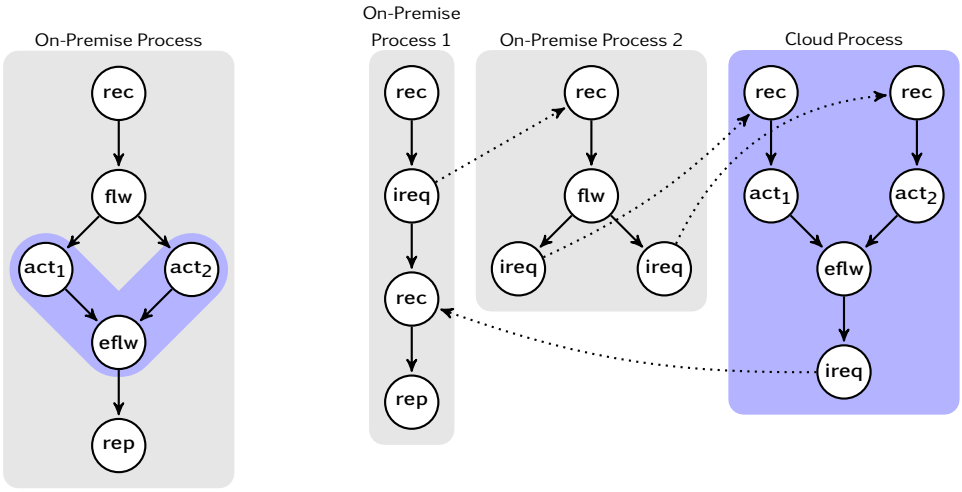
The following four situations are applicable:

- **Start node and branches distributed on-premise, end node distributed in the cloud**
  Figure 5.8 shows the situation in which the start node and the branches of the composite construct are marked for deployment in the cloud. The end-node needs to be situated on-premise. As a solution, the start node of the composite construct is placed together with the branches in a cloud process and the cloud process is invoked from on-premise process 1. Each of the branches of the composite construct



(a) situation (b) solution

Figure 5.8: Start node and branches distributed on-premise, end node distributed in the cloud



(a) situation (b) solution

Figure 5.9: Start node distributed on-premise, branches and end node distributed in the cloud

ends with an invocation to a second on-premise process. In this second on-premise process, the branches are joined. After the branches are joined, a notification is sent from the second to the first on-premise process to notify that execution of the composite construct has finished. The first on-premise process will reply to the invoker of the process.

Since on-premise process 1 and 2 both have the same distribution location, one could consider to place the reply node in the second on-premise process, instead of having to notify the first process. This, however, has consequences for the calling behavior of the process. The original process starts with a receive node and ends with a reply node. This indicates that the process will be executed synchronously and the invoker expects a reply from the process. In presented solution, the invoker will invoke on-premise process 1 synchronously, just as in the original situation. This means that on-premise process 1 needs a receive and a reply node. By moving the reply node to on-premise process 2, on-premise process 1 has no reply node anymore and invoking the process synchronously would not lead to a result. By introducing a callback from on-premise process 2 to on-premise process 1, the calling behavior of on-premise process 1 will conform to the calling behavior of the original process, since the reply node is placed in on-premise process 1.

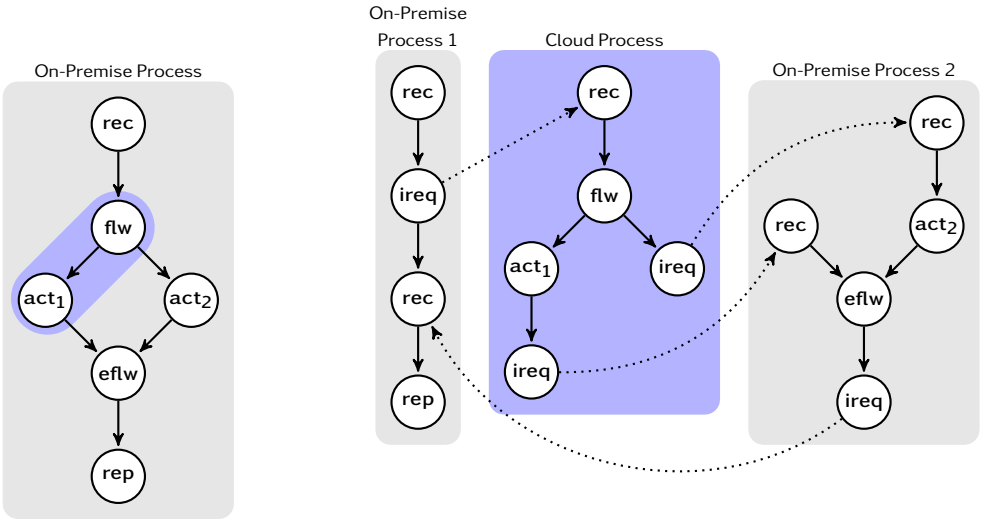
- **Start node distributed on-premise, branches and end node distributed in the cloud**
  Figure 5.9 shows the situation in which the start node is placed on-premise and activities in both branches are executed and joined in the cloud.

  To preserve the calling behavior of the original process, two on-premise processes are used. On-premise process 1 will be called and should have the same calling behavior as the original process. Therefore, the receive and reply node are situated in this process. The process will invoke on-premise process 2 and wait for an invocation from the cloud process, which indicates that the composite construct has been executed.

- **Start node and one branch distributed on-premise, other branch and end node distributed in the cloud**
  Figure 5.10 shows the situation in which the start node and one of the branches is placed in the cloud. The end node of the composite construct is placed on-premise. The second on-premise process invokes the first on-premise process to notify that the execution of the composite construct has finished. This invocation is introduced for the same reason as the earlier shown solutions in this category.

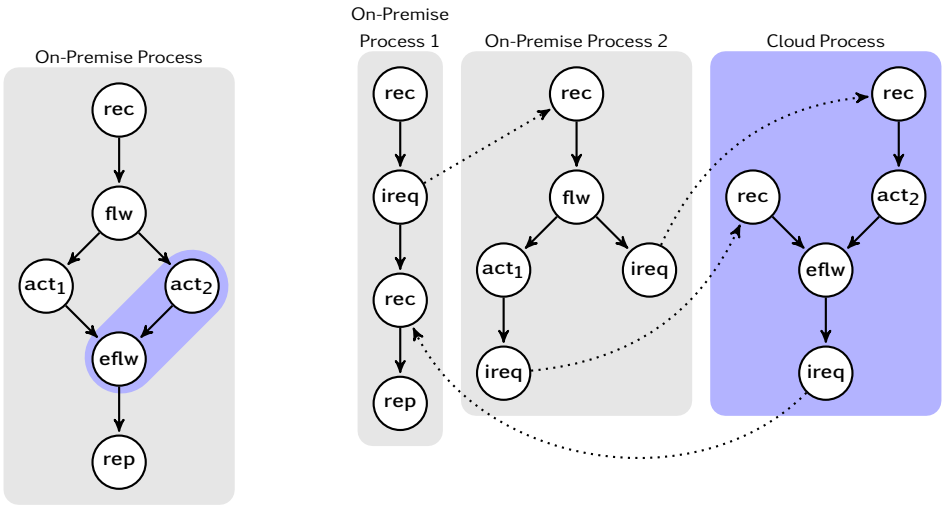
- **Start node and one branch distributed in the cloud, other branch and end node distributed on-premise**
  Figure 5.11 shows the situation in which one of the branches and the end node of a

composite construct are placed in the cloud. The call behavior of the original process is preserved by placing the receive and reply node of the original process in the first on-premise process.



(a) situation (b) solution

Figure 5.10: Start node and one branch distributed on-premise, other branch and end node distributed in the cloud



(a) situation (b) solution

Figure 5.11: Start node and one branch distributed in the cloud, other branch and end node distributed on-premise

5.4 Loops

Loop constructs in the intermediate model are categorized into two categories:

1. Loops in which the loop condition is evaluated before the execution of the loop branch.
2. Loops in which the loop condition is evaluated after the execution of the loop branch.

For both categories, we explain the possible decomposition solutions.

5.4.1 Loop with condition evaluation before branch execution

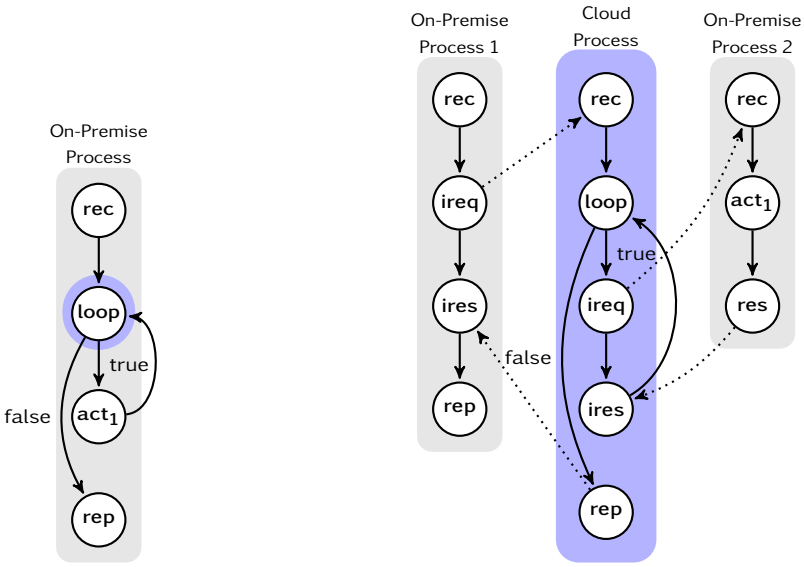
There are two situations possible when dealing with loop constructs in which the conditional node is evaluated before the execution of the loop branch.

- **Move construct as a whole**
  We omit the solution for this situation, since it is comparable to moving a composite construct as a whole, which is explained in section 5.3.1. The complete construct can be moved to a new process. The construct is replaced in the original process by synchronous invocation nodes.
- **Conditional node and nodes within loop branch are marked with different distribution locations**
  We can treat the loop branch within the loop construct as a separate process, since it is executed after a conditional node. A loop branch is only connected to the original process through the conditional node. Treating the branch as a separate process gives the opportunity to apply the other decomposition rules recursively on the branch. Figure 5.12 shows the situation in which the condition of a loop construct is moved to the cloud, whereas the activities within the loop branch are distributed in an on-premise process.

5.4.2 Loop with condition evaluation after branch execution

When dealing with loops in which the condition of the loop is evaluated after execution of the loop branch, two possible situations exist:

- **Move construct as a whole**
  This situation is comparable to moving a composite construct as a whole. We omit discussion of this situation since it is similar to the solution presented in section 5.3.1.

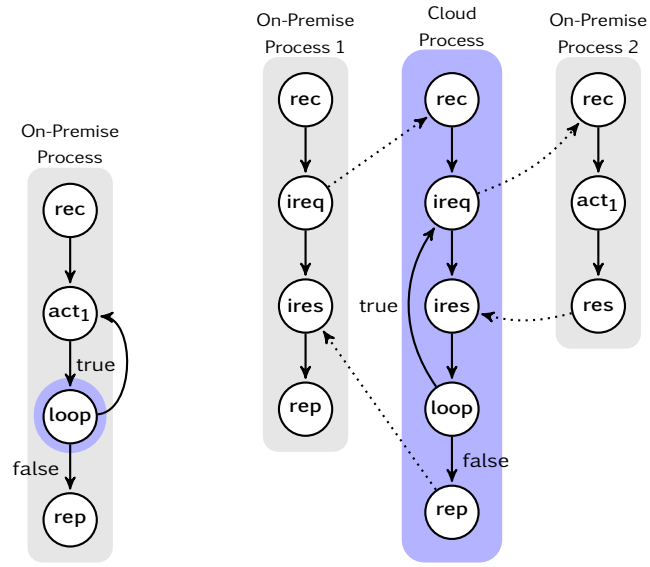


(a) situation (b) solution

Figure 5.12: Decomposition of a loop construct, with conditional node in the cloud and loop branch on-premise

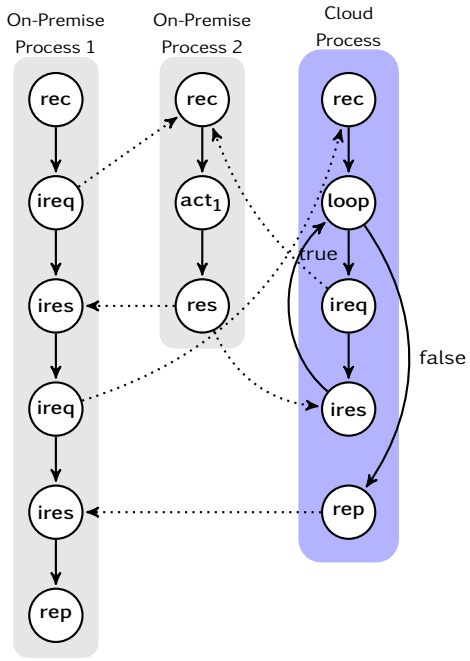
- **Conditional node and nodes within loop branch are marked with different distribution locations**
  There are two possible solutions for dealing with this situation. In the first solution, the loop branch and loop condition node are moved to a new process and are replaced in the original process by synchronous invocation nodes. In the newly created process (loop process), the loop branch is taken and moved to a separate process and called in the loop process by synchronous invocation nodes. This solution is shown in Figure 5.13b.

  The second solution is to move the loop branch to a separate process and rewrite the loop construct to a loop construct in which the condition is evaluated before the execution of the loop branch. The original process then first calls the loop branch, to execute the branch once before evaluation of the condition. After this invocation, a new invocation is used to call the loop process. In this loop process, the loop condition is evaluated first and depending on the result of the evaluation, the loop branch will be executed in which on-premise process 2 is invoked. This second solution is shown in Figure 5.13c.



(a) situation

(b) solution 1



(c) solution 2

Figure 5.13: Decomposition of a loop construct, with conditional node in the cloud and loop branch on-premise

5.5 Design decisions

Below, we discuss the design decisions that we took for implementing the decomposition transformation. These design decisions have been taken to simplify the implementation of the decomposition transformation. In a later stage, the decomposition algorithm can be extended to support more complex situations.

Process completeness

The input process for the transformation is restricted with the following constraints:

- The input process has at most one start node.
- The input process has at most one end node.
- The start-node of each composite construct should have a corresponding end-node, in which all of the branches of the composite construct are merged.

This decision was taken to avoid complex situations with multiple receive nodes at the beginning of a process or multiple reply nodes at the end of a process. In addition, the restrictions ensure that a process will never start or end with alternative or simultaneously executing branches.

Grouping sequential activities

Sequential activities with the same distribution location are always placed together in a process and are moved as a block to a new process, instead of being placed in separate processes. This decision was made to reduce the number of processes that will be generated during the decomposition transformation.

Keeping process together

When a sequence of activities is moved from one side to another, the surrounding process is kept together, as shown in solution 4 of the sequential activities, shown in Figure 5.3c. By keeping the process together, the original process will not be split and only new processes are generated for activities with a different distribution location than the original process. In the original process, these nodes are replaced by communication nodes. Since the structure of the process is maintained, the calling behavior of the process will also not change.

Branched activities treated as separate processes

Each branch within a composite construct is treated as a separate process. Nodes with the same distribution location as the surrounding composite construct stay within the branch of the construct. Nodes with a different distribution location are moved to separate processes. This decision gives us the opportunity to use the decomposition rules recursively on the branches of the composite constructs.

Composite construct start and end are distributed together

When dealing with composite constructs, we only allow the situation in which the composite construct is kept together. Different distribution locations for start and end nodes of composite constructs are not allowed. This decision was made to avoid complex situations with composite constructs. By keeping the start and end node together in the same process, we can treat them as block-structured elements [29] and perform the decomposition operations recursively on the branches of the construct.

Loop branches treated as separate processes

The branches of both types of loops are treated as separate processes. In case of loop constructs where the loop condition is evaluated after execution of the loop branch, we use the first solution, which is shown in Figure 5.13c. By treating the loop branch as a separate sub process we are able to use the decomposition rules recursively on the branch and treat the loop construct as a block-structured element.

6. Decomposition Implementation

This chapter discusses the implementation of the decomposition transformation. We decided to implement the transformation in two ways:

1. As graph transformation which was used for testing the steps of our algorithm. Details about the graph transformation implementation can be found in Appendix A. The initial decomposition transformation was created using graph transformations. The tool Groove [30] provided us with a graph transformation environment in which we could define transformation rules graphically. This gave us the opportunity to purely focus on the decomposition rules, without having to deal with the underlying data objects.

2. As a Java implementation of the algorithm. In this chapter we explain our Java solution, by using code fragments in pseudo code.

This chapter is structured as follows: Section 6.1 introduces the class diagram we have used in our Java implementation of the transformation. Section 6.2 introduces the transformation algorithm, by explaining each of the phases of the transformation. Section 6.3 to 6.6 discuss each transformation phase in detail. Section 6.7 discusses the verification algorithm we used for validating our solution.

6.1 Java classes

A simplified version of the class diagram we used in the implementation of our Java transformations is shown in Figure 6.1. We briefly explain each class below:

- **Graph**
  Graph is the main class for defining processes. A graph consists of a list of nodes, a list of edges, and a couple of functions for performing operations on the graph. The getAllGraphsAndSubGraphs function can be used for getting a list of graphs, in which the current graph is placed along with all the subgraphs that are available within the graph. Sub graphs are branches within composite constructs, such as loop branches or branches of a parallel/conditional constructs.

  The classes Graph, Node and Edge all have a hash map of attributes in which additional information about the objects can be stored.

- **Node**
  Node is the parent class for all the nodes. Each node has a unique name and distribution location, which indicates where the node should be located. The execu-

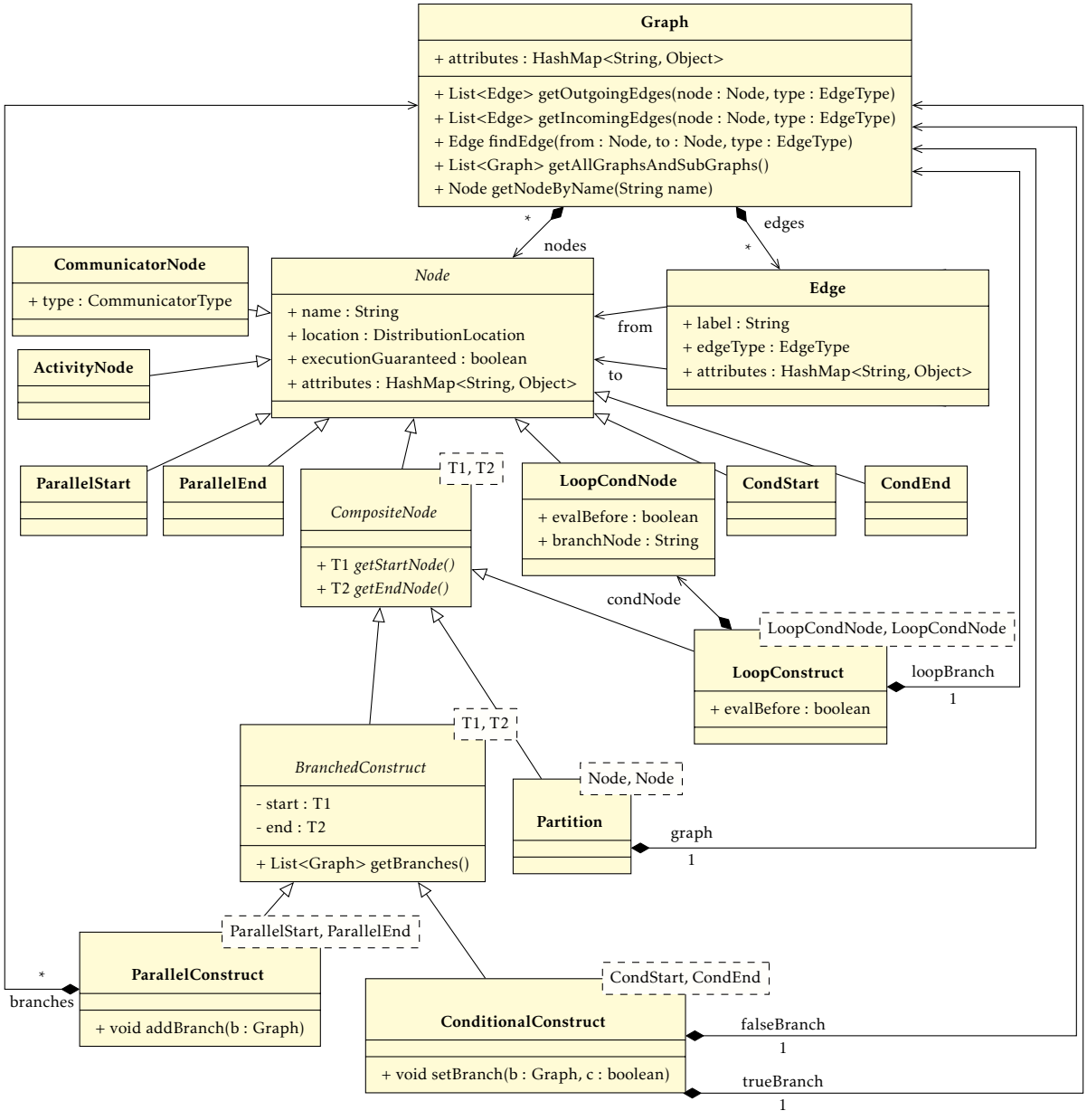


Figure 6.1: Class diagram of graph structure used in the Java implementation

tionGuaranteed attribute is used during the lifting transformation for optimizing the data dependency analysis.

- **Edge**
  An edge connects two nodes to each other. Each edge consists of a 'from' attribute, which represents the node from which the edge originates, and a 'to' attribute, which represents the node in which the edge terminates. Each edge has a specific edge type, which is either Control, Data or Communication. In addition, a label can be attached

to the edge by using the label attribute.

- **ActivityNode**
  Activity nodes are used to define activities within the process.
- **CommunicatorNode**
  A communicator is a node that communicates with another process. The communicator type of each communicator can be set by using the type attribute.
- **CompositeNode**
  Composite nodes consist of at least one subgraph. Each composite node has functions for getting the start and end nodes of the construct. These functions are implemented by each child of the CompositeNode. The template defined for CompositeNode is used for defining the type of the start and end node.
- **Partition**
  The Partition class is used to group adjacent nodes with the same distribution location.
- **BranchedConstruct**
  BranchedConstruct is a construct in which an execution branch is split up into multiple executing branches. We defined two subtypes of the construct for the time being: ParallelConstruct and ConditionalConstruct. A ParallelConstruct uses ParallelStart and ParallelEnd as respectively the start and end node for the construct, in case of a ConditionalConstruct, the CondStart and CondEnd nodes are used as start and end node, respectively.
- **LoopConstruct**
  LoopConstruct can be used for modeling loops. The loop construct has a reference to a LoopCondNode, which is the conditional node, and a loopBranch, which is the graph that is executed when the condition that is evaluated yields true. The evalBefore attribute is used for setting if the condition is evaluated before execution of the loop branch or afterwards.

Figure 6.2 shows the enumeration types that are used in the Java implementation of the transformations.

6.2 Transformations

The decomposition transformation transforms a process into multiple collaborating processes. The transformation consists of 4 phases. Instead of creating a new graph during

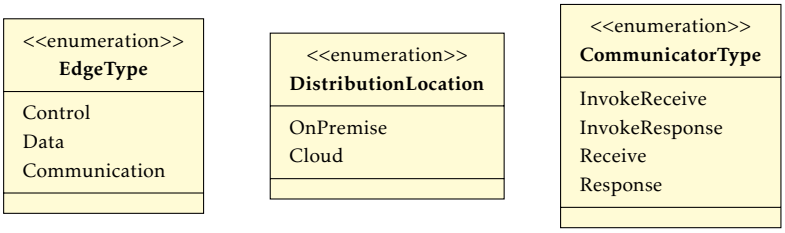


Figure 6.2: Enumeration types

each phase, the transformations modify the input graph. We briefly introduce these phases by explaining the goal, the input and the output of each phase:

Phase 1: Identification

Goal: Collect all the subgraphs, branched constructs and loop constructs that are nested in the graph, and mark each node with its desired distribution location. In addition, temporary nodes are added to the beginning of branches of branched constructs and loop constructs. These temporary nodes have the same distribution location as the surrounding construct and are necessary for correctly transforming the branches later on in the process.

Input:

- A graph that defines the original process.
- The activity distribution list.

Output:

- A list with all the subgraphs. These graphs represent process fragments, which are sub-processes of the original process.
- A list with all the branched constructs.
- A list with all the loop constructs.

Phase 2: Partitioning

Goal: Partition adjacent nodes with the same distribution location. These nodes should be placed together in one process and are therefore grouped together in a partition.

Input:

- The list with all the identified process fragments, from the previous phase.

Output:

- No output. Nodes within the process fragments are grouped in partitions.

Phase 3: Communicator node creation

Goal: Walks through all the graphs and creates communicators between partitions. The first two found partitions are examined by the algorithm. In both processes, communicator nodes are introduced and communication edges are added to the graph. The control edge that was present between the two partitions is deleted from the graph. If there is a third partition, the algorithm removes the edge between the second and third partition and merges the third partition with the first partition, since the third partition always has the same distribution location as the first partition. After the merge, the algorithm is repeated, until all the communicators are created between the partitions and there are no partitions left that can be merged.

Input:

- The list with all the partitioned graphs.

Output:

- No output. Adjustments are made to the inserted graphs.

Phase 4: Choreography creation

Goal: Remove the temporary nodes from the branched constructs and collect all the created processes, the communication edges and the data edges.

Input:

- The list with all the identified graphs.
- The list with all the branched constructs.
- The list with all the loop constructs.

Output:

- The initial graph on which the transformations are performed.
- A list with the communication edges.
- A list with all the data edges.

In the following sections we explain each of these phases in more detail, by giving their pseudo code. We use both procedures and functions. Procedures are functions that yield no result.

6.3 Identification phase

The input of the decomposition transformation is one single process and an activity distribution list. In this step, an algorithm examines the process and identifies composite constructs, loop constructs and branches within these constructs.

During the identification process, two additional tasks are performed:

1. Each of the nodes within the graph is marked with the distribution location of the node, as defined in the distribution list. In case of conditional and flow constructs, the distribution location of the start node of the construct is used as distribution location for both the start and the end node of the construct.
2. For each branch of a conditional or flow construct, and for each loop branch, a new temporary start node is added to the beginning of the branch. The temporary start node is marked with the distribution location of the start node of the composite construct. This node is used during the merging phase.

The algorithm is started by a call to the IdentifyProcessesAndMark procedure, with as parameters the start node of the input graph and the input graph itself. DistrLoc is a function that returns the desired distribution location for each node. Algorithm 1 shows the pseudo code for the identification phase.

6.4 Partitioning phase

During the partitioning phase, adjacent nodes with the same distribution location are allocated to the same partition. The algorithm is performed on each of the identified processes. The algorithm walks through each process fragment and compares the distribution location of a node with the distribution location of its successor node. When the distribution locations are the same, the nodes are placed in the same partition. Otherwise, both nodes are placed in different partitions, and a new control edge is created to connect the partitions to each other.

By applying this algorithm, the odd partitions will be merged, whereas the even partitions are separate processes. In a possible optimization phase, the even partitions can also be merged together.

Algorithm 1 Identification and marking algorithm

```
1:  BranchedConstructs ← {}
2:  LoopConstructs ← {}
3:  Processes ← {}

4:  procedure IdentifyProcessesAndMark(n, g)
5:      if n of type BranchedConstruct then
6:          BranchedConstructs ← BranchedConstructs ∪ {n}
7:          d ← distrLoc(n.start)
8:          n.location, n.start.location, n.end.location ← d        ▷ Mark with distribution location
9:          for all b ∈ n.getBranches() do
10:             if |b.nodes| > 0 then
11:                 WorkOnBranch(b, d)
12:             end if
13:         end for
14:     else if n of type LoopConstruct then
15:         LoopConstructs ← LoopConstructs ∪ {n}
16:         d ← distrLoc(n.condition)                               ▷ Mark with distribution location
17:         n.condition.location, n.location ← d                    ▷ Mark with distribution location
18:         WorkOnBranch(n.loopBranch, d)
19:     else
20:         n.location ← distrLoc(n)                                ▷ Mark with distribution location
21:     end if
22:     for all e ∈ g.getOutgoingEdges(n, Control) do               ▷ Follow outgoing edges
23:         IdentifyProcessesAndMark(e.to, g)
24:     end for
25: end procedure

26: procedure WorkOnBranch(branch,d)
27:     Processes ← Processes ∪ {branch}
28:     oldStart ← branch.start
29:     newNode ← new ActivityNode()                                ▷ Add temp start node
30:     newNode.location ← d                                        ▷ Mark with distribution location
31:     branch.nodes ← branch.nodes ∪ {newNode}
32:     branch.start ← newNode
33:     branch.edges ← branch.edges ∪ { new Edge(newNode, Control, oldStart)}
34:     IdentifyProcessesAndMark(oldStart, branch)
35: end procedure
```

The pseudo code of the partitioning algorithm is shown in Algorithm 2.

Example

A graphical example of the steps that are performed by the algorithm is shown in Figure 6.3. The dashed blocks around the nodes in the figure represent the partitions.

Algorithm 2 Partitioning algorithm

```
1:  procedure PartitionGraphs(Processes)
2:      for all g ∈ Processes do
3:          startNode ← g.start
4:          p ← new Partition()                          ▷ Create initial partition
5:          p.location ← startNode.location
6:          g.nodes ← g.nodes ∪ {p}
7:          g.start ← p
8:          partitionGraph(startNode, g, p)
9:      end for
10: end procedure

11: procedure PartitionGraph(n, g, p)
12:     p.graph.nodes ← p.graph.nodes ∪ {n}              ▷ Add node to partition
13:     g.nodes ← g.nodes − {n}                          ▷ Remove node from graph
14:     Edges ← g.getOutgoingEdges(n, Control)
15:     if |Edges| = 1 then
16:         e ← Edges.get(0)
17:         g.edges ← g.edges − {e}                      ▷ Remove edge from graph
18:         if e.to.location = n.location then           ▷ Following node in the same partition
19:             p.edges ← p.edges ∪ {e}
20:             PartitionGraph(e.to, g, p)
21:         else
22:             newPart ← new Partition()                ▷ Following node in a new partition
23:             newPart.location ← e.to.location
24:             g.nodes ← g.nodes ∪ {newPart}            ▷ Add new partition to graph
25:             g.edges ← g.edges ∪ { new Edge(p, Control, newPart)}   ▷ Create edge between partitions
26:             PartitionGraph(e.to, g, newPart)
27:         end if
28:     end if
29: end procedure
```

Activity 1 and 2 are marked for on-premise distribution and activity 3 and 4 are marked for being moved to the cloud (colored nodes), as shown in Figure 6.3a.

At first, a new partition is created and the first activity (act1) is added to the partition (p1), shown in Figure 6.3b. The successor node of activity 1 is examined. Since the successor node (act2) has the same distribution location as the current node (act1), the successor is added to the same partition as act1 and the control edge between the activities is also moved to the partition, as shown in Figure 6.3c.

The algorithm moves on, by looking at the successor of activity 2, which is activity 3 (act3). The distribution location of activity 3 is different than the distribution location of activity 2, which means that a new partition should be created for activity 3. A new partition (p2) is created and activity 3 is placed in this partition. The control edge between activity 2 and

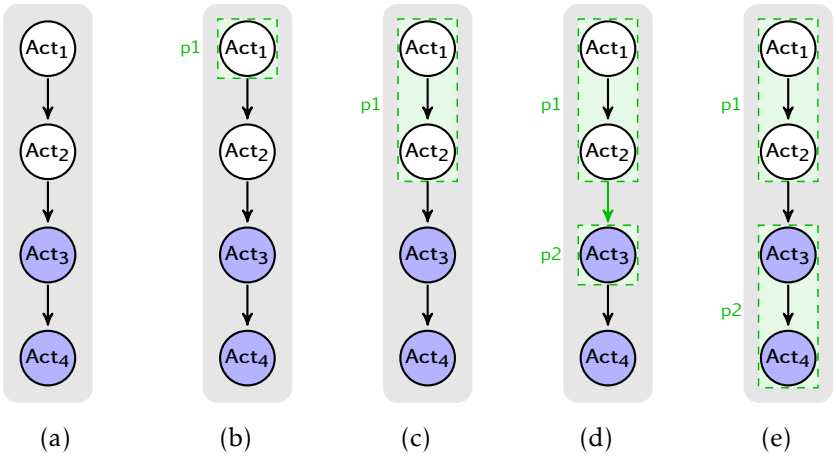


Figure 6.3: Example of the partitioning algorithm

activity 3 is removed, and a new control edge is created to connect the previous partition (p1) and the newly created partition (p2). This situation is shown in Figure 6.3d, where the colored arrow between p1 and p2 represents the newly created control edge.

The next step is to examine the successor of activity 3, which is activity 4 (act4). Activity 4 has the same distribution location as activity 3, which means that the node can be added to the same partition. The edge between the nodes is also moved to the partition. This situation is shown in Figure 6.3e. Since there are no nodes left in the process to examine, the algorithm terminates.

6.5 Communicator node creation phase

After the nodes in the processes are partitioned, communicators can be created between partitions. Communication between processes is implemented by using synchronous communication nodes. The algorithm takes the first partition of a process and identifies if there is a succeeding partition. If this is the case, communication nodes will be introduced at the end of the first partition for invoking the second partition. The second partition is delimited by a receive and a reply communicator node. The control edge that was present between the partitions is removed and replaced by communication edges. If there is a third partition, this partition should have the same distribution location as the first partition, since only two distribution locations are supported. The algorithm removes the control edge between the second and the third partition and merges the first and the third partition. The algorithm can now be repeated, until all communicators are created and no partitions can be merged anymore.

During the first phase of the decomposition algorithm, we introduced temporary nodes at

the beginning of each branch. The function of these nodes for the decomposition process should now become clear. When the decomposition algorithm is performed, the start node of the process determines where the process is deployed. Consider a parallel construct which is marked for being distributed on-premise, but the first activity within one of the branches has been marked for allocation in the cloud. Since all the branches are considered as being separate processes, the decomposition algorithm is performed on the branch and the algorithm thinks that the branch should be distributed in the cloud, whereas the surrounding construct is situated on-premise. By introducing a temporary node with the same distribution location as the surrounding construct to the beginning of the branch, the algorithm knows where the process should be distributed and creates correct communicators.

The algorithm is shown in Algorithm 3 as pseudo code.

Algorithm 3 Communicator creation algorithm

```
1: procedure CreateCommunicators(Processes)
2:     for all g ∈ Processes do
3:         startPartition ← g.start
4:         CreateCommunicators(startPartition, g)
5:     end for
6: end procedure
```

We will give a brief example of the algorithm on an example graph:

Example

Consider the partitioned process in Figure 6.4a. The algorithm starts by taking the first partition and adding two communicator nodes (ireq, ires) connected by a control edge at the end of the first partition. The second partition is placed in the cloud and surrounded with a receive and reply node. The first partition, which is extended with invocation nodes, is merged with the third partition (in which activity 4 is placed). This situation is shown in Figure 6.4b. The next step for the algorithm is to examine partition 1 again. Since there is a successor partition after partition 1, namely the partition in which activity 5 is placed, a communicator should be created. The partition in which activity 5 is placed is moved to a separate process and is surrounded with a receive and reply node. Invocation nodes are added at the end of the first partition, to invoke the newly created process. The algorithm can terminate now, since there are no other partitions succeeding partition 1.

```
 7: procedure CreateCommunicators(p, g)
 8:     if |g.getOutgoingEdges(p, Control)| = 1 then
 9:         p1p2Edge ← g.getOutgoingEdges(p, Control).get(0)
10:         p2 ← p1p2Edge.to
11:         g.edges ← g.edges − {p1p2Edge}
                                                    ▷ Create InvokeReceive Node
12:         invrecNode ← new CommunicatorNode(InvokeReceive, p.location)
13:         p.graph.edges ← p.graph.edges ∪ {newEdge(p.graph.end, Control, invrecNode)}
14:         p.graph.nodes ← p.graph.nodes ∪ {invrecNode}
                                                    ▷ Create Receive Node
15:         recNode ← new CommunicatorNode(Receive, p2.location)
16:         p2.graph.edges ← p2.graph.edges ∪ {newEdge(recNode, Control, p2.graph.start)}
17:         p2.graph.nodes ← p2.graph.nodes ∪ {recNode}
18:         p2.graph.start ← recNode
19:         g.edges ← g.edges ∪ {new Edge(invrecNode, Communication, recNode)}
                                                    ▷ Create Response Node
20:         resNode ← new CommunicatorNode(Response, p.location)
21:         p2.graph.edges ← p2.graph.edges ∪ {newEdge(p2.graph.end, Control, resNode)}
22:         p2.graph.nodes ← p2.graph.nodes ∪ {resNode}
                                                    ▷ Create InvRes Node
23:         invresNode ← new CommunicatorNode(InvokeResponse, p.location)
24:         p.graph.edges ← p.graph.edges ∪ {newEdge(invrecNode, Control, invresNode)}
25:         p.graph.nodes ← p.graph.nodes ∪ {invresNode}
26:         g.edges ← g.edges ∪ {newEdge(resNode, Communication, invresNode)}
                                                    ▷ Combine partition 1 and 3 (if available)
27:         if |g.getOutgoingEdges(p2, Control)| = 1 then
28:             p2p3Edge ← g.getOutgoingEdges(p2, Control).get(0)
29:             p3 ← p2p3Edge.to
30:             p.edges ← p.edges ∪ {newEdge(invresNode, Control, p3.graph.start)}
                                                    ▷ Copy nodes and edges to partition 1
31:             for all n ∈ p3.graph.nodes do
32:                 p.graph.nodes ← p.graph.nodes ∪ {n}
33:             end for
34:             for all e ∈ p3.graph.edges do
35:                 p.graph.edges ← p.graph.edges ∪ {e}
36:             end for
                                                    ▷ Update outgoing edges from partition 3
37:             for all e ∈ g.getOutgoingEdges(p3, Control) do
38:                 e.from ← p
39:             end for
                                                    ▷ Remove old edge and partition 2
40:             g.edges ← g.edges − {p2p3Edge}
41:             g.nodes ← g.nodes − {p3}
42:             CreateCommunicators(p, g)
43:         end if
44:     end if
45: end procedure
```

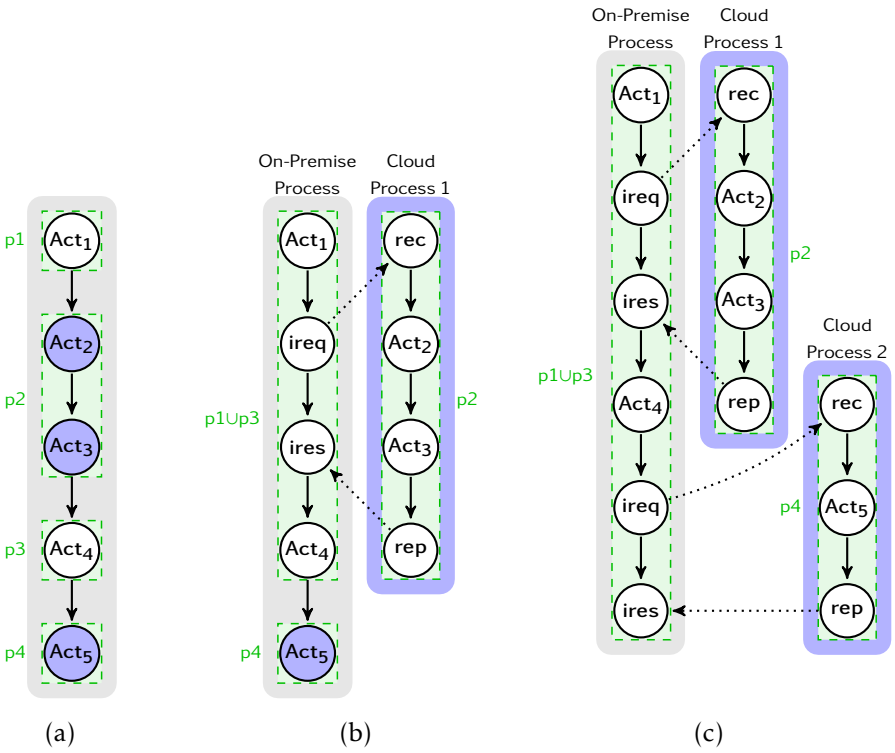


Figure 6.4: Example of the creating communicators algorithm

6.6 Choreography creation phase

In the last phase of the decomposition algorithm, all the created processes, communication edges and data edges are collected. The temporary nodes that were added to the beginning of the branches are removed. The processes, communication edges and data edges together form the choreography description for the decomposed business process.

After the previous phases, all the graphs consist of partitions, that are connected to each other by communication edges. Each partition is collected and is used as a separate process. The first partition in a process however, might be part of a composite construct and is therefore part of another process, namely the process in which the composite construct is placed. Therefore, the first step of this phase is to walk through the composite constructs and collect the created processes.

This algorithm consists of a couple of functions and procedures. The pseudo code of the algorithm is shown in Algorithm 4. We briefly explain below the functions and procedures that were used in the algorithm.

CollectOutputProcesses
The algorithm starts after a call to the CollectOutputProcess procedure. After the pro-

cedure has finished, the OutProc list is filled with all the separate processes, ComEdges list is filled with the communication edges between the processes and the DataEdges list contains the data edges.

The procedure first walks through the lists with branched constructs and the list with loop constructs and calls functions that deal with these constructs. After that, only the input process (the process that was used in the first phase) is left for examination. The graphs within the partitions of the input graph are copied to the output list and the communication and data edges are collected.

DealWithBranch
The DealWithBranch function performs operations on a branch of a branched construct or a branch of a loop construct. The function takes the first partition from the branch and removes the temporary start node from the graph within the partition. The branch of the construct is eventually replaced by this graph.

The next step is to copy the communication edges within the branch to the ComEdges list. The other partitions in the branch are copied to the OutProc list, since they represent newly created processes.

The function returns the inner graph from the first partition, which should be assigned as new branch graph in the construct.

6.7 Data dependency verification

Once the decomposition algorithm finishes, an algorithm for data verification is needed to check if no data restrictions have been violated as a result of the decomposition transformation. The algorithm we have implemented assumes that the process engine on which the process will be executed uses an execution strategy in which variables are used for passing data between activities. For example, engines that execute WS-BPEL [6] processes.

The last phase of the decomposition algorithm results in three lists: 1. list with all the created processes, 2. list with the communication edges between the processes and 3. list with all the data dependencies. These lists together form the input for the verification algorithm.

The Validate function walks through the list of all the data edges. For each data edge, the 'from' (n1) and 'to' node (n2) are selected. The label on the edge identifies the data item that is involved in the data dependency relation. The nodeInWhichGraph function is used to determine in which process the nodes are used. When the nodes are not in the same process, the findNode function is used to find the path that should be walked to get from n1 to n2. The nodes that were visited during the walk are collected in a list and represent

Algorithm 4 Choreography creation algorithm

```
 1: OutProc ← {}
 2: ComEdges ← {}
 3: DataEdges ← {}
 4: procedure CollectOutputProcesses(inputGraph, BranchedConstructs, LoopConstructs)
 5:     for all c ∈ BranchedConstructs do                ▷ Deal with branched constructs
 6:         if c of type ParallelConstruct then
 7:             ReplaceParallelNodeProcess(c)
 8:         else
 9:             ReplaceConditionalNodeProcess(c)
10:         end if
11:     end for
12:     for all c ∈ LoopConstructs do                    ▷ Deal with loop constructs
13:         ReplaceLoopNodeProcess(c)
14:     end for

15:     for all n ∈ inputGraph.nodes do                  ▷ Copy processes from the main graph
16:         OutProc ← OutProc ∪ {n.graph}
17:     end for
18:     for all e ∈ g.edges do                           ▷ Copy communication and data edges
19:         if e.type = EdgeType.Communication then
20:             ComEdges ← ComEdges ∪ {e}
21:         else
22:             DataEdges ← DataEdges ∪ {e}
23:         end if
24:     end for
25: end procedure
```

a walked path. After the path is found, the validatePath function is used to check if a data restriction is violated by the current data edge relation. The data restriction is violated whenever there is a node in the path list with a different distribution destination than the data restriction location for the current data item. The nodes that violate a certain data restriction are collected in a list and returned by the algorithm.

6.8 Conclusion

The phases presented in this chapter together form our solution for the decomposition transformation. By performing the phases on the subgraphs of the main graph, we can avoid complex situations, since all the nodes in each graph are treated during the transformation as single nodes (i.e. without looking at the type of the node or the possible contents within a node).

An optimization phase could be added to the decomposition transformation, to combine

```
26: procedure ReplaceParallelNodeProcess(c)
27:     branches ← {}
28:     for all g ∈ c.branches do
29:         branches ← branches ∪ DealWithBranch(g)
30:     end for
31:     c.branches ← branches
32: end procedure

33: procedure ReplaceConditionalNodeProcess(c)
34:     trueBranch ← null
35:     falseBranch ← null
36:     for all g ∈ c.branches do
37:         if |g.nodes| > 0 then
38:             if c.trueBranch = g then
39:                 trueBranch ← DealWithBranch(g)
40:             else
41:                 falseBranch ← DealWithBranch(g)
42:             end if
43:         end if
44:     end for
45:     c.trueBranch ← trueBranch
46:     c.falseBranch ← falseBranch
47: end procedure

48: procedure ReplaceLoopNodeProcess(c)
49:     c.loopBranch ← DealWithBranch(c.loopBranch)
50: end procedure

51: function DealWithBranch(branch)
                                        ▷ Restore graph by removing temporary nodes
52:     firstPart ← branch.start
53:     oldE ← firstPart.graph.getOutgoingEdges(firstPart.graph.start, Control).get(0)
54:     firstPart.graph.nodes ← firstPart.graph.nodes – {firstPart.graph.start}
55:     firstPart.graph.edges ← firstPart.graph.edges – {oldE}
56:     firstPart.graph.start ← oldE.to
57:     for all e ∈ branch.edges do                    ▷ Collect communication edges
58:         ComEdges ← ComEdges ∪ {e}
59:     end for
60:     for all n ∈ branch.nodes do                    ▷ Copy processes to output list
61:         if n ≠ branch.start then                   ▷ Avoid start partition, since it is no separate process
62:             OutProc ← OutProc ∪ {n.graph}
63:         end if
64:     end for
65:     return firstPart.graph
66: end function
```

Algorithm 5 Data restriction validation

```
1:  function VALIDATE(dataEdges, graphs)
2:      result ← new HashMap < String, List < Node >> ()
3:      for all e ∈ dataEdges do
4:          if e.from ≠ e.to then          ▷ Determine the graphs in which the nodes of the edge are defined
5:              startGraph ← nodeInWhichGraph(e.from)
6:              endGraph ← nodeInWhichGraph(e.to)
7:              dataItem ← e.label
8:              if startGraph ≠ endGraph then
9:                  path ← {}
10:                 if findNode(e.to, startGraph, path, {}) then          ▷ Find path from e.from to e.to
11:                     result[dataItem] ← result[dataItem] ∪ validatePath(e, path)          ▷ Validate path
12:                 end if
13:             else
14:                 result[dataItem] ← result[dataItem] ∪ validatePath(e, {})          ▷ Validate e.from and e.to
15:             end if
16:         end if
17:     end for
18:     return result
19: end function

20: function NODEINWHICHGRAPH(n, graphs)
21:     for all g ∈ graphs do
22:         for all sg ∈ g.getAllGraphsAndSubGraphs() do
23:             if n ∈ sg.nodes then
24:                 return g
25:             end if
26:         end for
27:     end for
28:     return null
29: end function
```

some of the newly created processes, to reduce the amount of data that is sent between processes. For example, in the communicator node creation phase odd partitions are merged, whereas even partitions are identified as new processes. A possible optimization would be to merge even partitions, based upon data dependency relations between nodes in partitions. Consider two even partitions (p2 and p4) with a data dependency between a node in p2 and a node in p4. When the partitions are separate processes, data needs to be sent from p2 to p4 via an intermediate odd partition (p1). By merging the partitions, data is directly available for the activity in p4 and therefore, no data needs to be send from p2 to p4. We implemented a simple version of this optimization, in which also the even partitions are merged. This solution was added to the Java implementation and can be selected by an extra input parameter, in which the partition merge type can be selected. In future work, a more advanced solution can be used in which data dependencies between partitions is

```
30: function findNode(nodeToFind, currentGraph, path, visitedGraphs, comEdges)
31:     if currentGraph ∈ visitedGraphs then                ▷ Ensure that no infinite loops occur
32:         return false
33:     end if
34:     visitedGraphs ← visitedGraphs ∪ {currentGraph}
35:     if nodeToFind ∈ currentGraph.nodes then
36:         return true
37:     end if
                                          ▷ Walk through invoking communicators in the process
38:     for all n ∈ getCommunicators(currentGraph, CommunicatorType.InvokeRec, CommunicatorType.Response)
    do
39:         correspondant ← getMatchingCommunicator(n, comEdges)      ▷ Get receiving communicator
40:         correspondantGraph ← nodeInWhichGraph(correspondant)
41:         newPath ← path ∪ {n} ∪ {correspondant}                          ▷ Copy path
42:         if findNode(nodeToFind, correspondantGraph, newPath, visitedGraphs) then
43:             path ← newPath                                      ▷ Copy the correct path
44:             return true
45:         end if
46:     end for
47:     return false
48: end function
```

used as basis for merging even partitions.

We verified the correctness of the decomposition solution informally by testing the solution on several business processes. For each of the obtained results we removed the communication nodes and replaced the communication edges with control edges, which resulted in the original processes again. This indicates that the behavior of the process itself is not changed by the transformation and no information from the original process is lost during the decomposition transformation. In future work, one could use formal verification to show the correctness of the rules.

```
49: function getCommunicators(process, comType)
50:     result ← {}
51:     for all g ∈ process.getGraphAndAllSubGraphs() do
52:         for all n ∈ g.nodes do
53:             if n of type CommunicatorNode ∧ n.type = comType then
54:                 result ← result ∪ {n}
55:             end if
56:         end for
57:     end for
58:     return result
59: end function

60: function getMatchingCommunicator(process, communicationEdges)
61:     for all e ∈ communicationEdges do
62:         if e.from = node then
63:             return e.to
64:         else if e.to = node then
65:             return e.from
66:         end if
67:     end for
68:     return result
69: end function

70: function validatePath(originalEdge, path)
71:     dataItem ← originalEdge.label
72:     distrLoc ← getDataItemRestriction(dataItem)
73:     result ← {}
74:     if distrLoc ≠ null then
75:         if originalEdge.from.location = distrLoc then
76:             result ← result ∪ {originalEdge.from}
77:         end if
78:         if originalEdge.to.location = distrLoc then
79:             result ← result ∪ {originalEdge.to}
80:         end if
81:         for all n ∈ path do
82:             if n.location ≠ distrLoc then
83:                 result ← result ∪ {n}
84:             end if
85:         end for
86:     end if
87:     return result
88: end function
```

7. Business Process Language Selection

In this chapter we give an overview of the business processes language we have used in the lifting and grounding phase. We decided to use Amber [24] as business process design language in this work.

This chapter is structured as follows: Section 7.1 explains the concepts supported by Amber. Section 7.2 discusses the mapping from the Amber concepts to the concepts of our intermediate model.

7.1 Amber

Amber [24] is a business processes design language which is used as the base language in BiZZdesigner. Amber has a graphical representation and is used for defining business processes in the design phase of the BPM lifecycle.

Amber considers 3 domains, namely the behavior domain for modeling business process behavior, the actor domain to model the participants involved in the processes, and the item domain to model the items that are used by the process. These domains are briefly introduced below.

7.1.1 Actor domain

The actor domain is used for describing organizations, departments, systems and people carrying out business processes. The main concept within the domain is the actor. An actor is a resource that performs a business process. In the actor domain, actors are structured elements which might contain other actors. Interaction points can be used to model relationships between actors, and between an actor and the environment the actor interacts with. A relation can consist of more than two interaction points in case multiple actors are involved in the interaction. An example of an actor model is shown in Figure 7.1.

7.1.2 Behavior domain

The behavior domain is used for describing the behavior of a business process. The main concept within this domain is action. An action represents an activity that is performed in the environment. Each action has two essential properties: 1. a property that represents the actor that performs the action, and 2. a property that represents the output of the action. Triggers are special types of actions that are executed immediately, and can therefore function as start nodes for a process. Actions are connected to each other by relations.

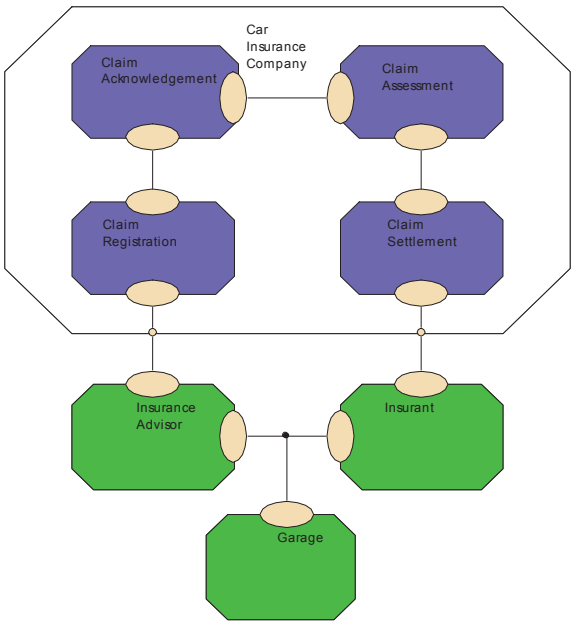


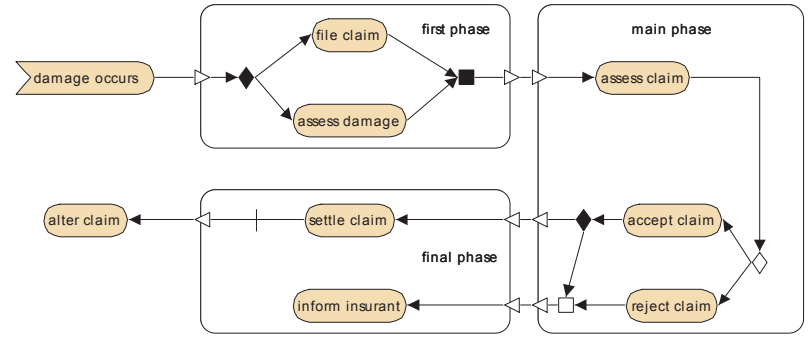
Figure 7.1: Example of actor model in Amber [24]

Relations between actions can be either enabling or disabling. Consider a process with actions a and b, and a relation between these actions. In case of an enabling relation, b can only be executed after a has finished. When a disabling relation is defined between a and b, b cannot happen any more after a has been executed.

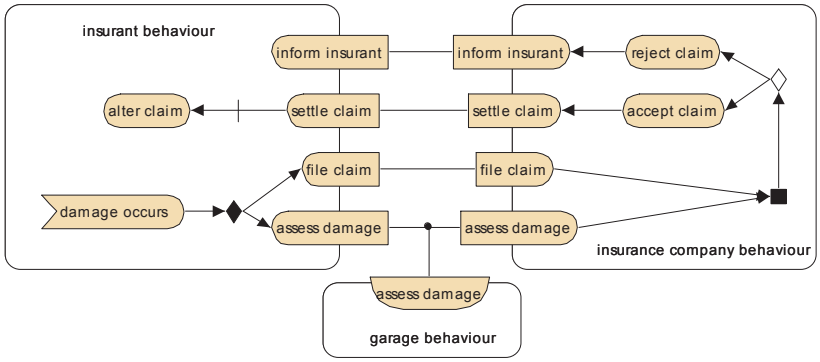
Parallel behavior and conditional behavior can be modeled by and-split/and-join and or-split/or-join nodes, respectively.

Behavior can be grouped in blocks. Blocks can be nested, just like actors in the actor domain. Blocks can be used for grouping behavior within a business process, but it is also possible to use blocks for describing the behavior of the participants in the process. In case part of a business process is defined in a block, the entry and exit points of the block are defined and can be connected to other activities in the process. In case of an interacting process, a block has interaction points that allow the interaction with the other participating processes.

An example of a behavioral model in Amber is shown in Figure 7.2. Figure 7.2a shows an example of a business process with behavior grouped in blocks, while Figure 7.2b shows the behavior of multiple interacting participants.



(a) Process with behavior grouped in blocks

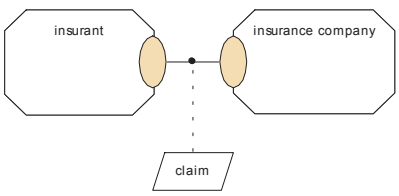


(b) Process with interaction between participants

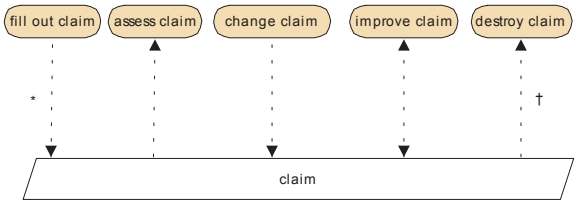
Figure 7.2: Example of behavior models in Amber [24]

7.1.3 Item domain

The item domain allows a process designer to describe the items that are handled in a business process. Items can be related to both the actor and the behavior domain. In the actor domain, items are coupled to interaction point relations, hereby indicating that the item is involved in the relation. In the behavior domain, items are coupled to actions or to interactions. An example of items in the actor domain is shown in Figure 7.3a, while usage of items in the behavior domain is shown in Figure 7.3b.



(a) Item in the actor domain



(b) Item in the behavior domain

Figure 7.3: Usage of items in Amber models [24]

7.1.4 BiZZdesigner

BiZZdesigner is a business process design tool built by BiZZDesign. The tool supports the Amber language, and all the concepts within the language can be defined in the tool. In addition, the tool offers support for writing scripts in order to perform complex operations on business processes.

7.2 Mappings

In order to understand how the lifting and grounding transformations should be defined, we need to identify how Amber elements can be mapped to our intermediate model.

As explained earlier, Amber consists of three domains. For our transformation, we only assess the behavior domain and the item domain, since these domains contain the information we need for our transformation. The behavior domain is used for identifying processes and the item domain is used for identifying data relations between activities and data items.

The following mappings are used from elements in Amber to constructs of the intermediate model:

Basic Concepts

Actions, Triggers and EndTriggers in Amber are mapped onto activity nodes in the intermediate model.

Conditional constructs

The Or-Split construct in Amber is mapped onto a conditional-node in the intermediate model. The Or-Join construct in Amber is mapped onto an end-contitional-node in the intermediate model. An example of conditional behavior in Amber is shown in Figure 7.4.

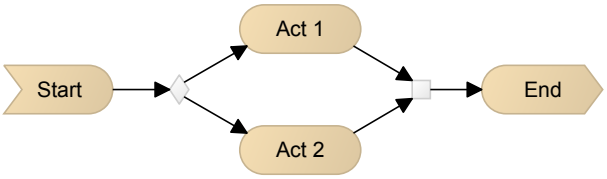


Figure 7.4: Conditional construct in Amber

Parallel constructs

The And-Split construct in Amber is mapped onto a flow-node in the intermediate model. The And-Join construct in Amber is mapped onto an end-flow-node in the

intermediate model. An example of parallel behavior in Amber is shown in Figure 7.5.

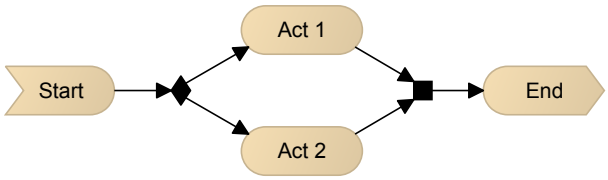


Figure 7.5: Parallel construct in Amber

Loops

Since the intermediate model is able to represent two possible types of loops, two loop mappings have been defined.

The situation in which the condition of a loop is evaluated before the execution of the loop branch is shown in Figure 7.6. In this situation, the or-split element in Amber is used as loop condition evaluator. This situation can be recognized by looking at the incoming edges of an or-split element. When the or-split has an incoming loop edge (represented by the double headed arrow) and in one of the branches, the actions are marked for repetitive behavior, then the or-split is used for defining a loop condition. No or-join node is needed, since the loop branch always points back to the or-split element.

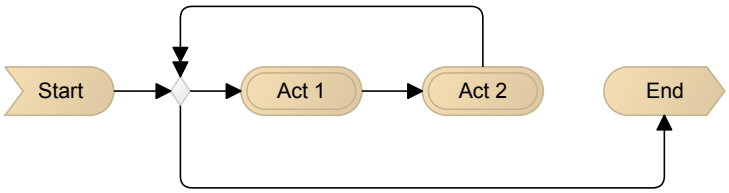


Figure 7.6: Condition evaluated before loop

The situation in which the condition of the loop branch is evaluated after execution of the loop branch is shown in Figure 7.7. Here, the activities in the loop branch are marked for repetitive behavior and the last activity in the branch points to an or-split. The or-split has one outgoing loop edge, which points to the first node of the loop branch.

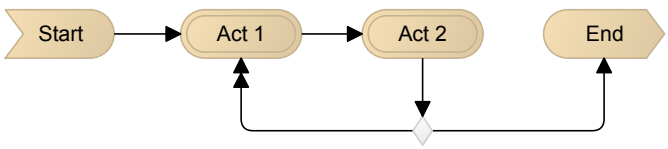


Figure 7.7: Condition evaluated after loop

Collaboration

After execution of the decomposition transformation, we have to consider the communication between processes. In Amber, processes and their interactions can be defined using subprocesses and interaction points. An example is shown in Figure 7.8.

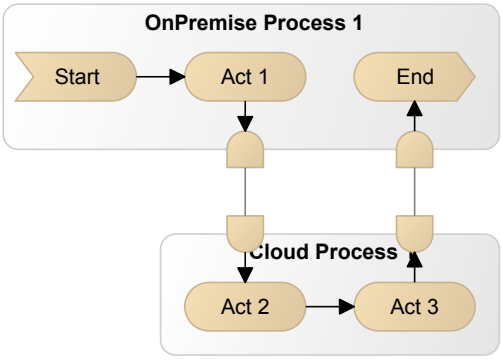


Figure 7.8: Interactions between two processes

Restriction

In order to simplify the lifting and grounding transformation, we restrict our mappings to the following behaviors:

- Each and-split element should have a corresponding and-join element.
- Each or-split element should have a corresponding or-join element. The only exception to this rule is when an or-split is used for defining loops. In this situation, the or-split element should have an incoming or outgoing loop edge.
- Only enabling relations (edges), item relations and loop relations are examined by the lifting algorithm.
- Subprocesses and constructs that are used in subprocesses, such as entry and exit point are avoided by the lifting algorithm. The grounding algorithm will create subprocesses for each generated process during the decomposition transformation.

In future work, more advanced concepts from Amber can be implemented in the lifting and grounding transformation.

8. Auxiliary transformations

This chapter discusses the implementation of the lifting and grounding transformations. Section 8.1 discusses the approach we have used to define our lifting transformation. Section 8.2 describes the model we use for exporting our business process from BiZZdesigner and importing it in Java. Section 8.3 describes the algorithm we use for replacing the parallel and conditional nodes by block structured parallel and conditional nodes. Section 8.4 explains how loop nodes are replaced by block structured constructs. Section 8.5 describes the algorithms we use to create data dependencies between the nodes in the business process. Section 8.6 discusses the grounding transformation.

8.1 Approach

Since we use Amber as base language and BiZZdesigner as process modeling tool, we would like to embed the complete transformation chain in BiZZdesigner, hereby giving users the opportunity to directly apply the transformation chain on their processes.

In order to support the marking of activities and data items for allocation in the cloud, we need to extend BiZZdesigner. In BiZZdesigner, profiles can be assigned to elements. New profiles are introduced for marking activities with their distribution location and data items with data restrictions. Two new profiles were added: OnPremise and Cloud. In case of activities and other elements in the behavior domain, only the Cloud flag can be set to the elements. When the flag is not set, the element is placed on-premise. In case of data items, both flags can be set. When both or none of the flags are set, the element can be moved without any restriction. When only one of the flags is set, the data should stay within the premises of the marked location.

The decomposition transformation has been written in Java. The first step is to convert our process defined in Amber to our intermediate model, which can be manipulated by our decomposition algorithm. We implemented an export script in BiZZdesigner for exporting the relevant information from an Amber process into XML format. Some alternatives are creating a Java parser for the BiZZdesigner file format, or a model-to-model transformation from the Amber metamodel to the intermediate model.

An XML parser is needed for reading the exported XML file and converting it to an instance of our intermediate model. The import of the XML file has been built in a Java project, which also performs some necessary steps on the intermediate model, such as data dependency analysis and replacement of composite structures.

Below we briefly introduce each of the phases of the lifting transformation. For each phase, we explain its goal, input and output.

1. **Export/Import phase**

   Goal: Export the business process from BiZZdesigner into a format that can be relatively easily imported by our Java implementation of the decomposition algorithm.

   Input: A business process defined in BiZZdesigner.

   Output: An XML representation of the business process, which can be imported by the decomposition algorithm.

2. **Parallel/Conditional block replacement phase**

   Goal: The conditional and parallel elements need to be replaced by conditional and parallel blocks. An algorithm is used for identifying these nodes and creating new block structured elements.

   Input: The imported graph.

   Output: The updated graph, with replaced conditional and parallel elements.

3. **Loop construct replacement phase**

   Goal: Find the loop conditional nodes and identify the loop branches that belong to these nodes. The found elements are converted into block structured loop constructs.

   Input: A graph in which the conditional and parallel elements already have been replaced by block structured elements.

   Output: An updated graph, with loop structured block elements.

4. **Data analysis**

   Goal: This phase consists of two steps: 1. analysis to determine whether execution is guaranteed. 2. creation of data dependencies between nodes.

   Input: The updated graph and a list with all the data relations between nodes and data items.

   Output: An updated graph, with data dependencies.

Below we discuss each of these phases in detail and describe the algorithms that we used for implementing the phases.

8.2 Export/Import

We use a simple XML representation for exporting the Amber process from BiZZdesigner and importing it in Java. The following XML format was used for describing a business process and the distribution list.

```
<?xml version="1.0" ?>
<orchestration>
  <nodes>
    <node id="unique Id" type="Trigger|EndTrigger|Action|And-Split|And-
        Join|Or-Split|Or-Join|LoopNode" name="String" branchNode="node.id"
        ? evaluateBefore="boolean"? />
  </nodes>
  <controlEdges>
    <controlEdge from="node.id" to="node.id" label="String"? />
  </controlEdges>
  <dataItems>
    <dataItem name="unique String" />
  </dataItems>
  <dataEdges>
    <dataEdge node="node.id" dataItem="dataItem.name" type="C|R|RW|W|D" /
        >
  </dataEdges>
  <distributionLocations>
    <distribution node="node.id" location="OnPremise|Cloud" />
  </distributionLocations>
  <dataRestrictions>
    <restriction dataItem="dataItem.name" location="OnPremise|Cloud" />
  </dataRestrictions>
</orchestration>
```

Listing 8.1: XML structure used for exporting an Amber business process

The XML structure consists of 6 sections:

- **Nodes**
  The Nodes sections contains the definition of all the nodes in the process. A unique identifier is assigned to each node, so that we can refer to them in other parts of the model. Each node is marked with its type. In case of an or-split that is used for evaluating a loop condition, the node is marked with the type LoopNode. The name of the node used in BiZZdesigner is stored in the name attribute.

  In the case of loops, two additional attributes are available: 1. the evaluateBefore attribute is set to true, in case the loopNode is evaluated before execution of the loop branch. 2. The loopBranch attribute points to the node in the loop branch, from which the loop edge originates, or in which the loop branch terminates, depending on the evaluation type of the loop.

  In the case of loop condition evaluation before loop branch execution, the loopNode attribute points to the end node of the loop branch. Otherwise, the loopNode attribute points to the start node of the loop branch. Loops are detected by searching for loop edges.

- **ControlEdges**
  The ControlEdges section consists of control edge definitions. For each control edge, the originator and destination needs to be set, which is done by filling in the 'from' and 'to' attribute respectively. Each attribute refers to a node id, which refers to a specific node. Additionally, a control edge can be labeled with a condition.
- **DataItems**
  The DataItems section is used for declaring the data items that are used within the process. For each data item one entry is added to the section, with the unique name of the data item as attribute.
- **DataEdges**
  The DataEdges section relates dataItems to nodes in the process. Each relation is described by using an dataEdge element, in which the node attribute refers to node id and dataItem refers to a data item. For each edge, the interaction type is defined. The following interaction types are available: Create (C), Read (R), Read-Write (RW), Write (W) and Destroy (D).
- **DistributionLocations**
  The DistributionLocations section consists of elements that relate node identifiers with their intended destination.
- **DataRestrictions**
  In this section, data items can be marked with a distribution location. A data restriction defines that a data item can only be used within the defined location. As soon as the data crosses the border of the location, the restriction is violated.

8.3 Parallel/Conditional block replacement

8.3.1 Analysis

Figure 8.1 shows the class diagram for the parallel and conditional constructs. Both classes inherit from the abstract class BranchedConstruct. Templates are used for defining the specific types of the start and end nodes of the constructs. The name and location attribute are taken from the Node class, which is shown in the full class diagram in Figure 6.1.

The ParallelConstruct and ConditionalConstruct classes are block structured elements. The original start and end node of the construct are stored in the start and end attribute, respectively. The branches of the constructs are stored either in a branches list, in case of a ParallelConstruct, or in the variables trueBranch and falseBranch, in case of a ConditionalConstruct.

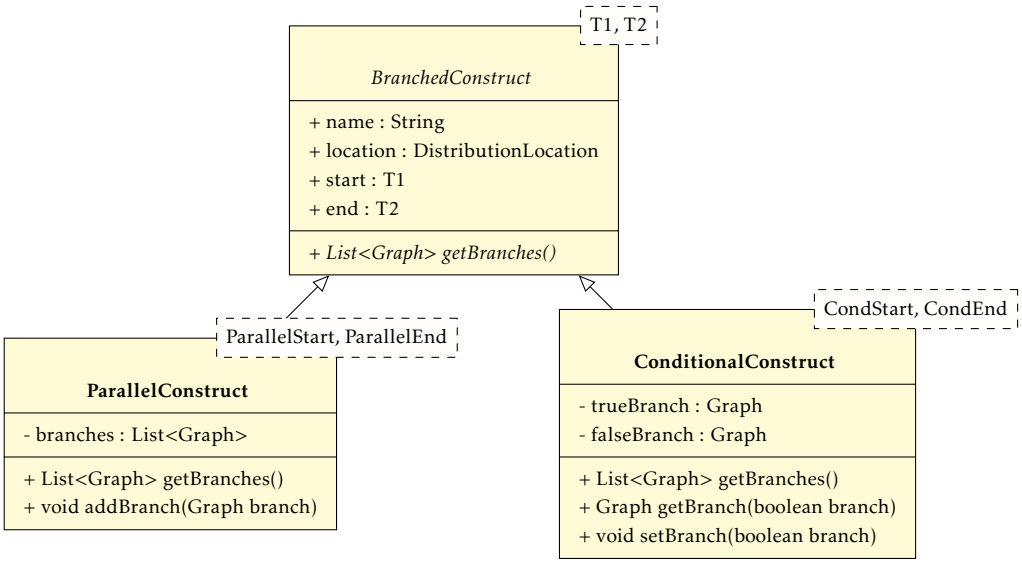
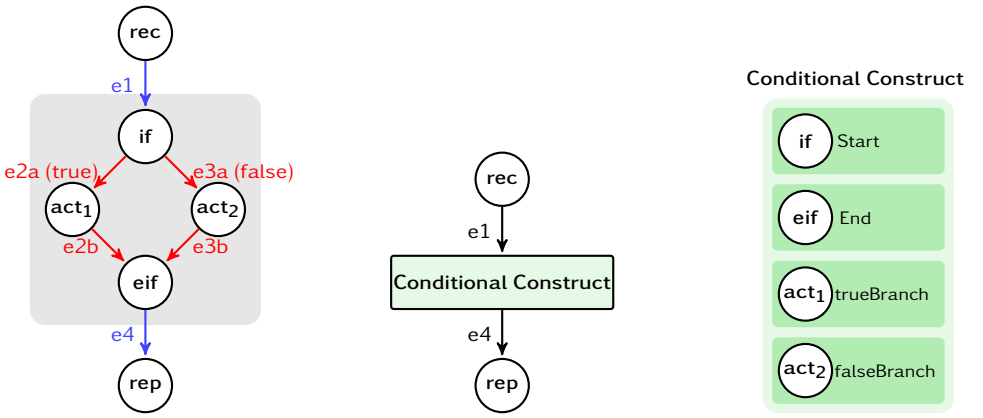


Figure 8.1: Class diagram for branched constructs

Figure 8.2a shows an example of a process with conditional nodes. The part of the process that is changed and replaced with a conditional construct is represented by the marked area in the figure. A unique number has been assigned to each edge. The edges (e1 and e4) are edges that need to be updated. After the replacement, edge e1 should point to the conditional construct instead of pointing to the if-node and edge e4 should originate from the conditional construct instead of originating from the eif-node. The other edges (e2a, e2b, e3a, e3b) need to be removed from the graph.

Figure 8.2b shows the graph, after the conditional nodes are replaced with a conditional construct. The contents of the conditional construct are shown in Figure 8.2c.



(a) Before replacement (b) After replacement (c) Contents of block

Figure 8.2: Creation of branched constructs

8.3.2 Algorithm

The first step after importing the intermediate model is to match composite constructs and replace the nodes by a new instance of a composite construct. The algorithm starts by calling the StartCompositeNodeReplacement function, with the source graph as input. The result of the function is the input graph, with replaced composite constructs. An outline of the function is shown in Algorithm 6.

Algorithm 6 StartCompositeNodeReplacement

```
1: function StartCompositeNodeReplacement(graph)
2:     ReplaceCompositeNodes(graph.getStartNode(), graph, graph, {})
3:     return graph
4: end function
```

ReplaceCompositeNodes is the main function, which is used for walking through the graph and identifying conditional and parallel nodes. The pseudo code of the function is shown in Algorithm 7. The function consists of the following steps:

1. Check if the current node that is examined by the algorithm has been processed before. Since there might be loops in the graph, we have to ensure that the algorithm does not iterate infinitely. When a node has already been visited, the function returns null.
2. When the current node is a start node of a conditional construct, a new conditional construct should be generated. This is done by invoking the ReplaceConditionalNode function. The current node is changed to the generated construct.
3. When the current node is a start node of a parallel construct, a new parallel construct should be generated. The function ReplaceParallelNode is called to perform this operation. The current node is changed to the generated construct.
4. When the current node is an end node of a composite construct, the algorithm returns the current node.
5. In case of other types of nodes, the node is copied to the newly generated graph. The node is added to the visited list.
6. The outgoing edges of the current node are examined and for each outgoing edge, the ReplaceCompositeNodes function is called.

The ReplaceConditionalNodes function creates ConditionalConstruct nodes. The pseudo code of the function is shown in Algorithm 8. The function consists of the following steps:

1. A new conditional construct node is created and the original conditional start node is set as start node within the construct.

Algorithm 7 ReplaceCompositeNodes

```
 1: function ReplaceCompositeNodes(n, newGraph, lookupGraph, Visited)
 2:     if n ∈ Visited then                                              ▷ Step 1
 3:         return null
 4:     else if n of type ConditionalStartNode then                      ▷ Step 2
 5:         n ← ReplaceConditionalNodes(n, newGraph, lookupGraph, Visited)
 6:     else if n of type ParallelStartNode then                         ▷ Step 3
 7:         n ← ReplaceParallelNodes(n, newGraph, lookupGraph, Visited)
 8:     else if n of type ConditionalEndNode ∨ n of type ParallelEndNode then  ▷ Step 4
 9:         return n
10:     else                                                             ▷ Step 5
11:         newGraph.nodes ← newGraph.nodes ∪ {n}
12:         Visited ← Visited ∪ {n}
13:     end if
14:     for all e ∈ lookupGraph.getOutgoingEdges(n, Control) do          ▷ Step 6
15:         newGraph.edges ← newGraph.edges ∪ {e}
16:         o ← ReplaceCompositeNodes(e.to, newGraph, lookupGraph, Visited)
17:         if o ≠ null then
18:             result ← o
19:         end if
20:     end for
21:     return result
22: end function
```

2. The outgoing edges of the conditional start nodes are examined, and for each outgoing edge a new branch graph is created. The ReplaceCompositeNodes function is used for walking through the branches and for adding the nodes to the newly created branches. The ReplaceCompositeNodes algorithm returns the conditional end node, which belongs to the conditional construct. After the branch is created and the end node is discovered, the edges from the last node in the branch towards the conditional end node are removed from the branch graph. The branch is set either as true or false branch in the conditional construct, depending on the label of the edge, between the conditional start node and the first node in the branch.

3. A check is needed to ensure that the algorithm has found a correct end node for the construct. When no end node is found, or when a parallel end node is found instead of a conditional end node, the algorithm yields an exception, since the input graph is not valid.

4. The end node is set as end node in the conditional construct and the construct is added to the newly created graph.

5. The incoming edges of the conditional start node need to be updated, since they need to point to the conditional construct at this point.

Algorithm 8 ReplaceConditionalNodes

```
 1: function ReplaceConditionalNodes(n, newGraph, lookupGraph, Visited)
 2:     construct ← new ConditionalConstruct()                          ▷ Step 1
 3:     construct.start ← n
 4:     endNode ← null
 5:     for all e ∈ lookupGraph.getOutgoingEdges(n, Control) do         ▷ Step 2
 6:         branch ← new Graph()
 7:         endNode ← ReplaceCompositeNodes(e.to, branch, lookupGraph, Visited)
 8:         if e.label = true then
 9:             construct.trueBranch ← branch
10:         else
11:             construct.falseBranch ← branch
12:         end if
13:         for all oldE ∈ branch.getIncomingEdges(endNode, Control) do
14:             branch.edges ← branch.edges − {oldE}
15:         end for
16:     end for
17:     if endNode = null ∨ endNode not of type ConditionalEndNode then  ▷ Step 3
18:         throw Exception "No valid end node found. The graph is invalid."
19:     else
20:         construct.end ← endNode                                     ▷ Step 4
21:         newGraph.nodes ← newGraph ∪ {construct}
22:         for all e ∈ lookupGraph.getIncomingEdges(n, Control) do     ▷ Step 5
23:             e.to ← construct
24:         end for
25:         for all e ∈ lookupGraph.getOutgoingEdges(endNode, Control) do ▷ Step 6
26:             e.from ← construct
27:             if e.to ∈ Visited then
28:                 newGraph.edges ← newGraph.edges ∪ {e}
29:             end if
30:         end for
31:     end if
32:     Visited ← Visited ∪ {endNode}
33:     return construct
34: end function
```

6. The outgoing edges of the conditional end node need to be updated. The conditional construct should be selected as the originator of each of these edges.

The ReplaceParallelNodes function is almost the same as the ReplaceConditionalNodes function. Instead of having to determine if a branch is either the true branch or the false branch, the function just adds the newly created branch to the branches list within the newly created construct. By taking the code of ReplaceConditionalNodes and replacing step 2 by the pseudo code, shown in Algorithm 9, the ReplaceParallelNodes function is obtained.

Algorithm 9 ReplaceParallelNodes (partially)

```
1:  function ReplaceParallelNodes(n, newGraph, lookupGraph, Visited)
2:      ...
3:      for all e ∈ lookupGraph.getOutgoingEdges(n, Control) do          ▷ Step 2
4:          branch ← new Graph()
5:          endNode ← ReplaceCompositeNodes(e.to, branch, lookupGraph, Visited)
6:          construct.branches ← construct.branches ∪ {branch}
7:          for all oldE ∈ branch.getIncomingEdges(endNode, Control) do
8:              branch.edges ← branch.edges − {oldE}
9:          end for
10:     end for
11:     ...
12: end function
```

8.4 Loop construct replacement

8.4.1 Analysis

Figure 8.3 shows a class diagram fragment for using loop constructs. The LoopConstruct class consists of a condNode attribute, in which the conditional loop node will be stored. The branchNode attribute is filled during the export/import phase and points to the node that was present in the Amber process on the other side of the loop edge. The evaluateBefore attribute indicates whether the loop condition is evaluated before or after execution of the loop branch and the loopBranch attribute is a graph, which contains the activities that are executed in the loopbranch.

Figure 8.4a shows a process with a loop. The marked part will be replaced by a loop block. The blue edges (e1 and e5) need to be updated after the replacement. Edge e1 needs to point to the loop block after replacement and edge e5 needs to originate from the loop block. The result after the replacement is shown in Figure 8.4b. Figure 8.4c shows parts of the contents of the loop construct. The edges e4 and e3 have been removed and the loop branch is separated from the conditional node.

8.4.2 Algorithm

The loop construct replacement algorithm consists of several functions. For both types of loops, two different approaches are needed. The algorithm is started by calling the StartLoopNodeReplacement function, with the graph as input parameter. In this graph, the parallel and conditional nodes should already have be replaced by block structured nodes. The algorithm maintains a queue of graphs that need to be examined by the algorithm. For each graph, a function is called to get the graph and all the subgraphs within

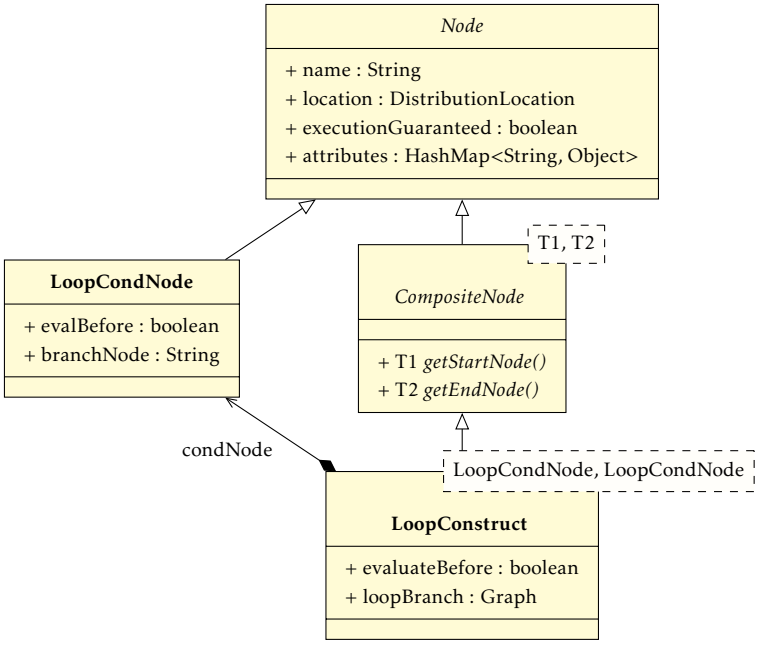
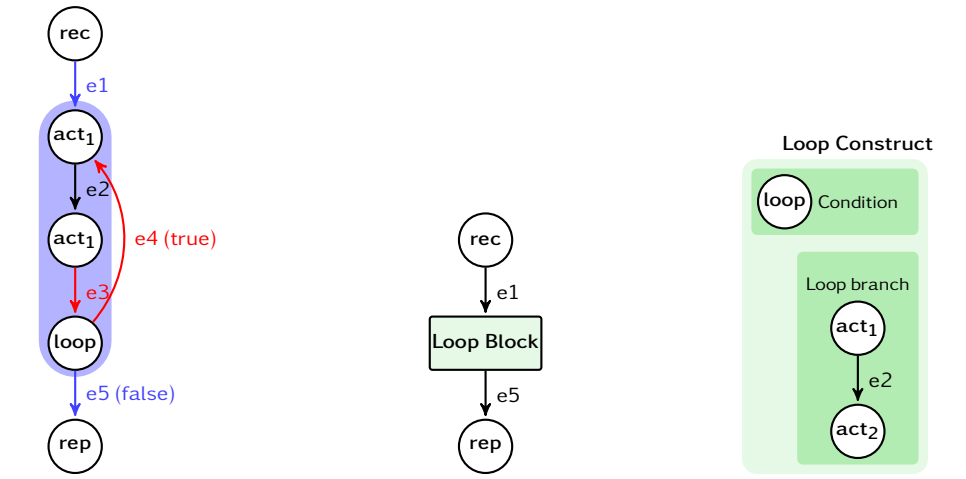


Figure 8.3: Class diagram for loop constructs



(a) Before replacement (b) After replacement (c) Contents of block

Figure 8.4: Creation of loop constructs

the graph. Subgraphs are graphs that are located in composite constructs within the graph. For example, the branches of a conditional construct are subgraphs of the graph in which the conditional construct is located.

For each graph, the algorithm will call the FindLoopNodes function, which searches for instances of the LoopCondNode class within the nodes list of a graph. These nodes are created during the export/import phase and represent loop condition nodes. When a Loop-

CondNode is found, the algorithm will check if the node will be evaluated before or after the loop branch. Depending on this decision, either the createConstructWithPreEvaluation function or the createConstructWithPostEvaluation function is called.

Algorithm 10 Loop construct identification

```
1:  graphQueue ← {}

2:  function StartLoopNodeReplacement(graph)
3:      graphQueue ← newQueue(graph.getAllGraphsAndSubGraphs())
4:      while !graphQueue.isEmpty() do
5:          g ← graphQueue.take()
6:          loopNodes ← FindLoopNodes(g)
7:          if |loopNodes| > 0 then
8:              loopNode ← loopNodes.first()
9:              if loopNode.evaluateBefore then
10:                 createConstructWithPreEvaluation(g, loopNode)
11:             else
12:                 createConstructWithPostEvaluation(g, loopNode)
13:             end if
14:         end if
15:     end while
16:     return graph
17: end function

18: function FindLoopNodes(g)
19:     result ← {}
20:     for all n ∈ g.nodes do
21:         if n of type LoopConditionalNode then result ← result ∪ {n}
22:         end if
23:     end for
24:     return result
25: end function
```

Loop construct with evaluation of condition after branch execution

Creation of a loop construct in which the loop branch is executed after the evaluation of the loop condition is the simplest case, since the branchNode attribute of the LoopCondNode points to the start node of the loop branch. Algorithm 11 shows the outline of the algorithm. The createLoopConstruct function is used to create the loop construct. After the loop construct is created and inserted in the graph as replacement for the nodes involved in the loop, the algorithm will add the current graph to the queue again. This is necessary, since there might be more loop nodes in the process. In addition, the loop branch of the newly created construct is added to the queue, to search for nested loops.

Algorithm 11 CreateConstructWithPostEvaluation

```
26: procedure createConstructWithPostEvaluation(g, loopNode)
27:     branchStartNode ← loopNode.branchNode
28:     construct ← createLoopConstruct(g, loopNode, branchStartNode)
29:     graphqueue ← graphqueue ∪ {construct.graph} ∪ {g}
30: end procedure
```

The createLoopConstruct function creates a new instance of a LoopConstruct. The conditional node is set and the copyUntilConditionalNodeFound function is used for walking through the loop branch and moving the nodes and edges from the original graph to the loop branch graph. The code for this function is shown in Algorithm 13. After the loop branch is created, the conditional node needs to be removed from the input graph and the loop edge, between the conditional node and the start node of the loop branch needs to be deleted from the input graph. The last step is to update edges in the input graph.

Algorithm 12 CreateLoopConstruct

```
31: function createLoopConstruct(g, loopNode, branchStartNode)
32:     construct ← new LoopConstruct()                          ▷ Create construct
33:     construct.condNode ← loopNode
34:     construct.evaluateBefore ← loopNode.evaluateBefore
35:     g.nodes ← g.nodes ∪ {construct}
                                                                 ▷ Create loop branch
36:     copyUntilConditionalNodeFound(branchStartNode, loopNode, construct.loopBranch, g, {})
                                                                 ▷ Remove old node/edge
37:     g.edges ← g.edges − {g.findEdge(loopNode, branchStartNode, EdgeType.Control)}
38:     g.nodes ← g.nodes − {loopNode}
                                                                 ▷ Update old edges
39:     for all e ∈ g.getIncomingEdges(branchStartNode, EdgeType.Control) do
40:         e.to ← construct
41:     end for
42:     for all e ∈ g.getIncomingEdges(loopNode, EdgeType.Control) do
43:         e.to ← construct
44:     end for
45:     for all e ∈ g.getOutgoingEdges(loopNode, EdgeType.Control) do
46:         e.from ← construct
47:     end for
48: end function
```

Loop construct with evaluation of condition before branch execution

The function CreateConstructWithPreEvaluation, shown in Algorithm 15, deals with loops in which the loop condition is evaluated before execution of the loop branch. The first step of the function is to find the start node of the loop branch. Since the loop edge, shown in

Algorithm 13 CopyUntilConditionalNodeFound

```
49: procedure copyUntilConditionalNodeFound(curNode, condNode, newGraph, lookupGraph, visited)
50:     if curNode ∈ visited then                          ▷ Security check to avoid infinite loops
51:         return
52:     else
53:         visited ← visited ∪ {curNode}
54:     end if
                                          ▷ Remove node from lookupGraph and add to newGraph
55:     lookupGraph.nodes ← lookupGraph.nodes − {curNode}
56:     newGraph.nodes ← newGraph.nodes ∪ {curNode}
57:     for all e ∈ g.getOutgoingEdges(curNode, EdgeType.Control) do
58:         lookupGraph.edges ← lookupGraph.edges − {e}
59:         if e.to ≠ condNode then
60:             newGraph.edges ← newGraph.edges ∪ {e}
61:             copyUntilConditionalNodeFound(e.to, condNode, newGraph, lookupGraph, visited)
62:         end if
63:     end for
64: end procedure
```

Figure 7.6, points from the last node in the loop branch to the loop condition node. By examining the outgoing edges from the loop condition and walking the paths until the end node of the branch is found, we can determine which of the paths is the loop branch. The isLoopBranch function is used for checking which of the paths is the loop branch. The code of the algorithm is shown in Algorithm 14.

After the start node of the loop branch has been found, the CreateLoopConstruct is used to create the construct and copy the nodes within the loop branch from the original graph to the loop branch graph inside the loop construct.

8.5 Data analysis

After the intermediate model is created, data analysis needs to be performed to discover the data dependencies between nodes. During the export/import phase, data edges were created, which relate a data item to a node. Each data edge contains a type, which indicates if a node creates, reads, writes or destroys the data.

Our data analysis algorithm consists of two phases. At first, for each node in the intermediate model the boolean property executionGuaranteed is set. This property indicates if the node is always executed in the model, or if an execution is possible in which the node is not executed. This information helps us to reduce the number of data dependencies between nodes.

The second phase create data dependencies between the nodes in the intermediate model

Algorithm 14 IsLoopBranch

```
65: function isLoopBranch(curNode, branchEndNode, g, visited)
66:     if curNode ∈ visited then
67:         return false
68:     else
69:         visited ← visited ∪ curNode
70:     end if
71:     if curNode = branchEndNode then
72:         return true
73:     end if
74:     for all e ∈ g.getOutgoingEdges(curNode, EdgeType.Control) do
75:         if isLoopBranch(e.to, branchEndNode, g, visitedNodes) then
76:             return true
77:         end if
78:     end for
79:     return false
80: end function
```

that share the same data items.

Below, we will describe both the algorithms we use for data analysis.

8.5.1 Mark execution guarantees

The first step of the data analysis is to set the executionGuarantee property for every node in the process. The property indicates if the execution of the node is guaranteed, in case of normal behavior, i.e. behavior in which the process is completely executed without being interrupted because of failure.

Execution of nodes is not guaranteed when:

1. A node is placed in a branch of a conditional construct. Conditional constructs evaluate a condition, to decide which of the outgoing branches needs to be executed. Since the condition is evaluated at runtime, execution of activities in a branch cannot be guaranteed.

2. Nodes are placed in the branch of a loop construct, where the loop condition is evaluated before the execution of the loop branch. Since the loop condition may yield false immediately, the execution of the loop branch is not guaranteed

Our algorithm for marking nodes with execution guarantees is shown in Algorithm 16. The following steps are taken in the algorithm:

- **Step 1:** The executionGuaranteed attribute is set for the current node

Algorithm 15 CreateConstructWithPreEvaluation

```
81: procedure createConstructWithPreEvaluation(g, loopNode)
82:     branchEndNode ← loopNode.branchNode
83:     branchStartNode ← null
                                          ▷ Find the start node of the loop branch
84:     outgoingEdges ← g.getOutgoingEdges(loopNode, EdgeType.Control)
85:     if |outgoingEdges| = 0 ∨ |outgoingEdges| > 2 then
86:         throw exception "Invalid number of outgoing edges for loop conditional node"
87:     else if |outgoingEdges| = 1 then
88:         branchStartNode ← outgoingEdges.get(0).to
89:         n1IsLoopBranch ← isLoopBranch(branchStartNode, branchEndNode, g, {loopNode})
90:         if !n1IsLoopBranch then
91:             throw exception "No loop branch found"
92:         end if
93:     else
94:         n1 ← outgoingEdges.get(0).to
95:         n1IsLoopBranch ← isLoopBranch(n1, branchEndNode, g, {loopNode})
96:         n2 ← outgoingEdges.get(1).to
97:         n2IsLoopBranch ← isLoopBranch(n2, branchEndNode, g, {loopNode})
98:         if (n1IsLoopBranch ∧ n2IsLoopBranch) ∨ (!n1IsLoopBranch∧!n2IsLoopBranch) then
99:             throw exception "No loop branch found"
100:        end if
101:        branchStartNode ← (n1IsLoopBranch)?n1 : n2
102:    end if
                          ▷ Create the loop construct and add graphs to the queue
103:    construct ← createLoopConstruct(g, loopNode, branchStartNode)
104:    graphQueue ← graphQueue ∪ {construct.loopBranch} ∪ {g}
105: end procedure
```

- **Step 2:** When the current node is a BranchedConstruct (either a ConditionalConstruct or a ParallelConstruct), the executionGuaranteed attribute of the start node and end node of the construct is set. For each of the branches of the construct, the procedure is called recursively. When the current node is a ConditionalConstruct, the execution of the nodes within the branches is not guaranteed and thus, the guaranteed parameter of the function should be changed to false.

- **Step 3:** In case of a LoopConstruct, the executionGuaranteed attribute needs to be set for the loop condition node. The executionGuaranteed attribute value for each of the nodes within the loop branch depends on if the loop condition is evaluated before or after execution of the loop branch.

- **Step 4:** The outgoing edges of the current node are followed.

Algorithm 16 MarkExecutionGuarantees

```
 1: procedure MarkExecutionGuarantees(n, g, guaranteed)
 2:     n.executionGuaranteed ← guaranteed                                  ▷ Step 1
 3:     if n of type BranchedConstruct then                                 ▷ Step 2
 4:         n.start.executionGuaranteed, n.end.executionGuaranteed ← guaranteed
 5:         for all branch ∈ n.branches do
 6:             if n of type ConditionalConstruct then
 7:                 MarkExecutionGuarantees(branch.start, branch, false)
 8:             else
 9:                 MarkExecutionGuarantees(branch.start, branch, guaranteed)
10:             end if
11:         end for
12:     else if n of type LoopConstruct then                                ▷ Step 3
13:         n.conditionalNode.executionGuaranteed ← guaranteed
14:         if n.evaluateConditionBefore then
15:             MarkExecutionGuarantees(n.loopbranch.start, n.loopbranch, false)
16:         else
17:             MarkExecutionGuarantees(n.loopbranch.start, n.loopbranch, guaranteed)
18:         end if
19:     end if
20:     for all e ∈ g.getOutgoingEdges(n, Control) do                       ▷ Step 4
21:         MarkExecutionGuarantees(e.to, g, guaranteed)
22:     end for
23: end procedure
```

8.5.2 Create data dependencies

During the import phase, data edges were created. A data edge relates a node to a dataitem together with a type. The type indicates whether the node creates, reads, writes or destroys the data item. The goal of the data dependency analysis algorithm is to walk through the whole graph once for each data item and collect the possible writers for the data item during the walk. As soon as a node (n) is reached, that uses the data item, a data dependency edge will be created from each of the possible writers, to n. The algorithm is based upon the data analysis idea in [20], in which the possible writer sets for nodes are collected. We separated the algorithm in a couple of procedures, to be able to better explain the algorithm.

The AnalyzeDependencies procedure walks through the list with all the declared data items. For each of the data items, the data edges in which the data item is used are selected and collected in a hashmap, where the key is the node involved and the value is the data edge. The CreateDataDepencies algorithm will be called to walk through the graph.

The CreateDataDependencies procedure consists of 4 steps:

- **Step 1:**

Algorithm 17 Data dependency analysis

```
1:  procedure AnalyzeDependencies(graph, dataItems, dataEdges)
2:      for all dataItem ∈ dataItems do
3:          curEdges ← {}
4:          for all e ∈ dataEdges do
5:              if e.dataItem = dataItem then
6:                  curEdges ← curEdges ∪ {e.node => e}
7:              end if
8:          end for
9:          CreateDataDependencies(graph.start, graph, graph, curEdges, {})
10:     end for
11: end procedure
```

The possible writer set is updated first. This is done by using the UpdatePossibleWriter function. If the currently examined node is defined in the data edges hashmap, the type of the data edge is considered. Depending on this type the following actions are taken:

- **Create:** The list with possible writers is extended with the current node. A data dependency edge will be created from the current node pointing to itself, to define the creation of a data item.
- **Read:** Data dependency edges will be created from each of the possible writers to the current node.
- **Write or ReadWrite:** Data dependency edges will be created from each of the possible writers to the current node. If the execution of the node is guaranteed, the possible writers list will be cleared. Nodes that will be examined later that read the data item only need a data dependency edge to the last writer. Since the execution of the node is guaranteed, only the last writer is a possible writer for the node. The last step is to add the node to the possible writers set.
- **Destroy:** Data dependency edges will be created from each of the possible writers to the current node. The possible writers list will be cleared, since the item is destroyed.

- **Step 2:** If the current node is a LoopConstruct, the algorithm will be called recursively on the loop branch.
- **Step 3:** If the current node is a BranchedConstruct, the algorithm will be called recursively on each of the branches. During the execution of the algorithm on the branches, changes to the possible writer list are monitored. When new possible writers are added, they are collected in a separate list. After the algorithm has finished for each branch, the algorithm will check if new writers have been selected. In this

situation, the old writer list is cleared and filled with the newly found writers.

- **Step 4:** The outgoing edges of the current node are followed and the algorithm continues to the next node.

```
12: procedure CreateDataDependencies(n, graph, lookupGraph, dataEdges, posWriters)
13:     UpdatePossibleWriters(n, graph, lookupGraph, dataEdges, posWriters)        ▷ Step 1
14:     if n of type LoopConstruct then                                            ▷ Step 2
15:         CreateDataDependencies(n.loopBranch.start, n.loopBranch, lookupGraph, dataEdges, posWriters)
16:     else if n of type BranchedConstruct then                                   ▷ Step 3
17:         newPosWriters ← {}
18:         newWritersFound ← false
19:         for all branch ∈ n.branches do
20:             posWritersBranch ← posWriters
21:             CreateDataDependencies(branch.start, graph, lookupGraph, dataEdges, posWritersBranch)
22:             if posWritersBranch ≠ posWriters then
23:                 newPosWriters ← newPosWriters ∪ posWritersBranch
24:                 newWritersFound ← true
25:             end if
26:         end for
27:         if newWritersFound then
28:             posWriters ← newPossibleWriters
29:         end if
30:     end if
31:     for all e ∈ graph.getOutgoingEdges(n, Control) do                          ▷ Step 4
32:         CreateDataDependencies(e.to, graph, lookupGraph, dataEdges, posWriters)
33:     end for
34: end procedure
```

8.6 Grounding

The previous sections described all the actions that were needed for lifting the a BiZZdesigner model to an instance of the intermediate model. In this section we look at the grounding transformation, in which the lists that were created during the decomposition transformation are used to obtain a new BiZZdesigner model.

8.6.1 Export/Import

The outcome of the decomposition transformation were 3 lists: a list with all the processes, a list with all the communication edges between those processes and a list with all the data edges. During the export phase, data is read from these lists and converted into an XML representation, which in turn is imported by a script in BiZZDesigner. The following XML format is used:

```
35: procedure UpdatePossibleWriters(n, graph, lookupGraph, dataEdges, posWriters)
36:     if n ∈ dataEdges.keys then
37:         de ← dataEdges.getValueForKey(n)
38:         switch de.type do
39:             case Create
40:                 posWriters ← posWriters ∪ {n}
41:                 CreateDataEdge(lookupGraph, n, n, de)
42:             case Read
43:                 CreateDataEdges(lookupGraph, posWriters, n, de)
44:             case ReadWrite ∨ Write
45:                 CreateDataEdges(lookupGraph, posWriters, n, de)
46:                 if n.executionGuaranteed then
47:                     posWriters ← {}
48:                 end if
49:                 posWriters ← posWriters ∪ {n}
50:             case Destroy
51:                 CreateDataEdges(lookupGraph, posWriters, n, de)
52:                 posWriters ← {}
53:         end switch
54:     end if
55: end procedure

56: procedure CreateDataEdges(g, writers, n, dataEdge)
57:     for all writer ∈ writers do
58:         CreateDataEdge(g, writer, n, dataEdge)
59:     end for
60: end procedure

61: procedure CreateDataEdge(g, fromNode, toNode, dataEdge)
62:     e ← new Edge(fromNode, toNode, Data)
63:     e.label ← dataEdge.dataItem
64:     g.edges ← g.edges ∪ {e}
65: end procedure
```

```
<?xml version="1.0" ?>
<choreography>
  <dataItems>
    <dataItem name="unique String" distributionLocation="OnPremise|Cloud"
        restrictionViolated="boolean"? />
  </dataItems>
  <orchestrations>
    <orchestration name="String" distributionLocation="OnPremise|Cloud">
      <nodes>
        <node id="unique Id" type="Trigger|EndTrigger|Action|And-Split|
            And-Join|Or-Split|Or-Join|Interaction"
          name="String" repeated="boolean"? restrictionViolated="boolean"
              ? />
      </nodes>
      <edges>
        <edge from="node.id" to="node.id" type="Control|Loop" label="
            String"? />
      </edges>
    </orchestration>
  </orchestrations>
  <interactionEdges>
    <edge from="node.id" to="node.id" type="Control" />
  </interactionEdges>
  <dataEdges>
    <edge from="node.id" to="dataItem.name" type="Data" relationType="C|R
        |RW|W|D" />
  </dataEdges>
</choreography>
```

Listing 8.2: XML structure used for importing the choreography in BiZZdesigner

The XML file represents a choreography description and consists of 4 parts:

- **Data items**
  The dataItems section contains the data items that were used in the process. Each data item has a unique name and a distribution location, which is the restriction which was set for the data item. The restrictionViolated attribute is used for setting if the data restriction was violated during the decomposition.

- **Orchestrations**
  In the orchestrations section, each of the processes that was created is described. For each process the nodes and edges are defined. Each node has a name, an Amber type, and a unique id. The attribute restrictionViolated is set when this node violates a data item restriction and the repeated boolean is set, if the activity is part of a loop branch.

  Edges consist of a reference to an originator node and a terminator node and a type, which is either a control in case of a control edge, or loop in case of a loop edge. The label attribute can be used to assign a label to the edge.

- **Interaction Edges**
  This section contains edges between processes, so called communication edges.

- **Data Edges**
  In this section, nodes are connected to the data items they use. For each data edge,

the interaction type is set, using the relationType attribute.

8.6.2 BiZZdesigner script restrictions

The script language within BiZZdesigner has a couple of restrictions. In this section we explain which restrictions we faced and which alternatives we used.

- **Interaction relation**
  For each process, a new sub process is created in the behavioural model. Interactions between those subprocesses should be performed by using interaction edges between interaction elements. BiZZdesigners script engine however, does not allow interaction edge creation. Therefore, we decided to create entry and exit elements for each interaction element and connect those elements with each other, using normal enabling relations.
- **Automatic layout**
  BiZZdesigner has no automatic layout mechanism. The created script will therefore just place the created elements on the canvas and users need to layout the diagram manually.

9. Case study

In this chapter we perform our transformation chain on a real-life example, the talent show audition. Below, the case is explained and for each of the phases in the transformation chain, intermediate results are shown.

9.1 Talent show audition process

Consider that a television broadcast company wants to produce a new singing competition show. The company uses an on-line registration system, in which contestants can register for the show. In order to get selected for the show, contestants need to upload an audition video, in which they are performing a song, and some personal information, so that producers can contact them when the contestant is selected for the show. The selection procedure of the contestants consists of two parts. At first, producers and a jury look at all the videos and directly select contestants for the show. The other video auditions are placed on the website of the show and visitors of the website can vote on the videos they like. The highest voted video auditions are selected and added to the contestants.

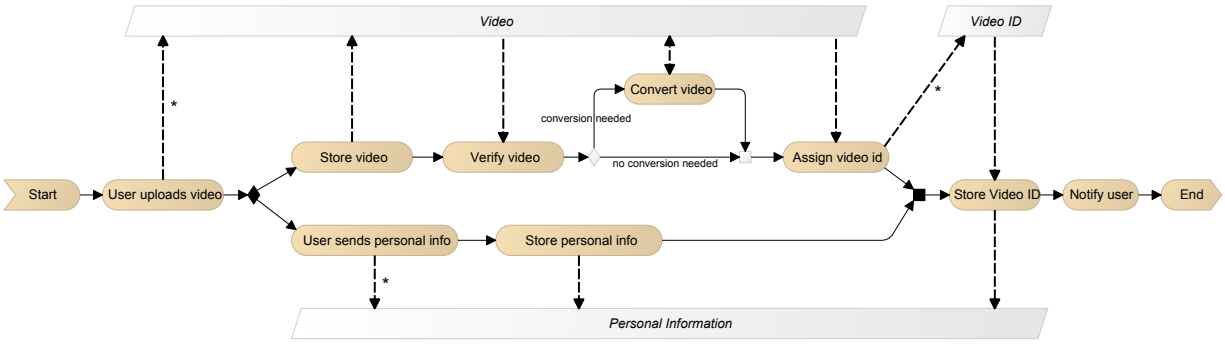


Figure 9.1: Business process of the on-line registration system

The business process of the on-line registration system is shown in Figure 9.1. The first step for the user is to upload a video. After the video is uploaded, the process is split up into two separate simultaneously executing branches:

1. The first branch performs operations on the uploaded video. At first, the video is stored in a folder on the server. After that, a verification algorithm is used to check if the video is valid. This operation also determines the videos properties, such as the format, size and quality. For the producers it is important that the videos are all in the same format, in order to speed up their selection process. In addition, when the videos need to be placed on the website, one video format is also needed. Therefore, a conversion step is needed. The conversion step is only performed when

the video does not comply to the selected format yet. After conversion, a unique video identifier is assigned to the video.

2. The second branch waits for personal information that should be submitted by the user. When the information is provided, the personal information is stored in a database.

After the branches merge again, the video identifier should be stored within the database of personal information. This step is necessary to know which video audition belongs to which personal information. After the personal information is updated, a notification is sent to the user and the business process is terminated.

9.2 Marking activities and data items

The television broadcast company expects a large amount of auditions. Since the storage of videos might take a lot of space, and operations on the videos, such as video conversion, are computation-intensive operations, the company has decided to make use of cloud computing for storing the videos and performing operations on the videos. The personal information of the process, however, should stay within the premises of the television broadcast organization. Therefore, the business process mainly runs on the server on-premise, while parts of the process are outsourced to the cloud.

Figure 9.2 shows the business process, marked with distribution locations and data restrictions. The activities that should be performed in the cloud are marked with the cloud flag. In Figure Figure 9.2, these activities are marked with a dark background color. The personal information data item is marked with a data restriction, which states that the item should stay on-premise. This is shown in Figure 9.2 by the shaded data item.

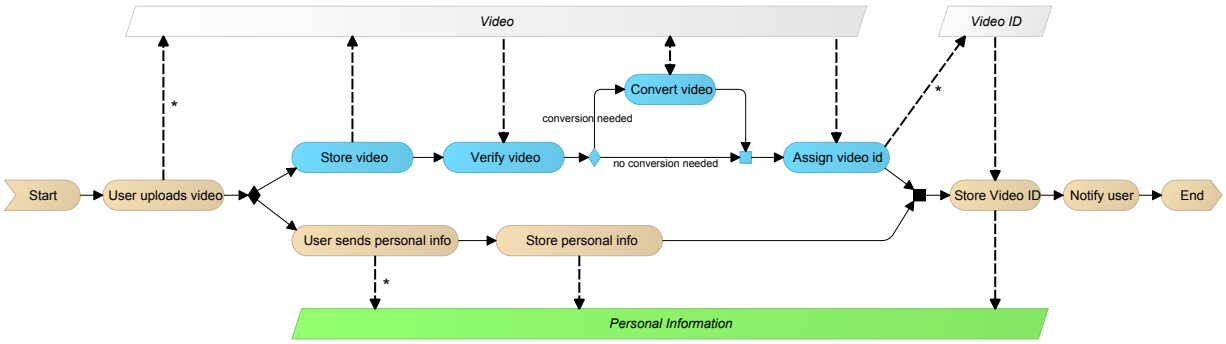


Figure 9.2: Business process with marked activities and data restrictions

9.3 Lifting

Once activities have been marked with a distribution location, and a data restriction has been placed on the Personal Information data item, the transformation chain can be started. At first, the lifting transformation is performed. Below we show three intermediate results which have been generated by our Java implementation during the lifting transformation.

9.3.1 Export

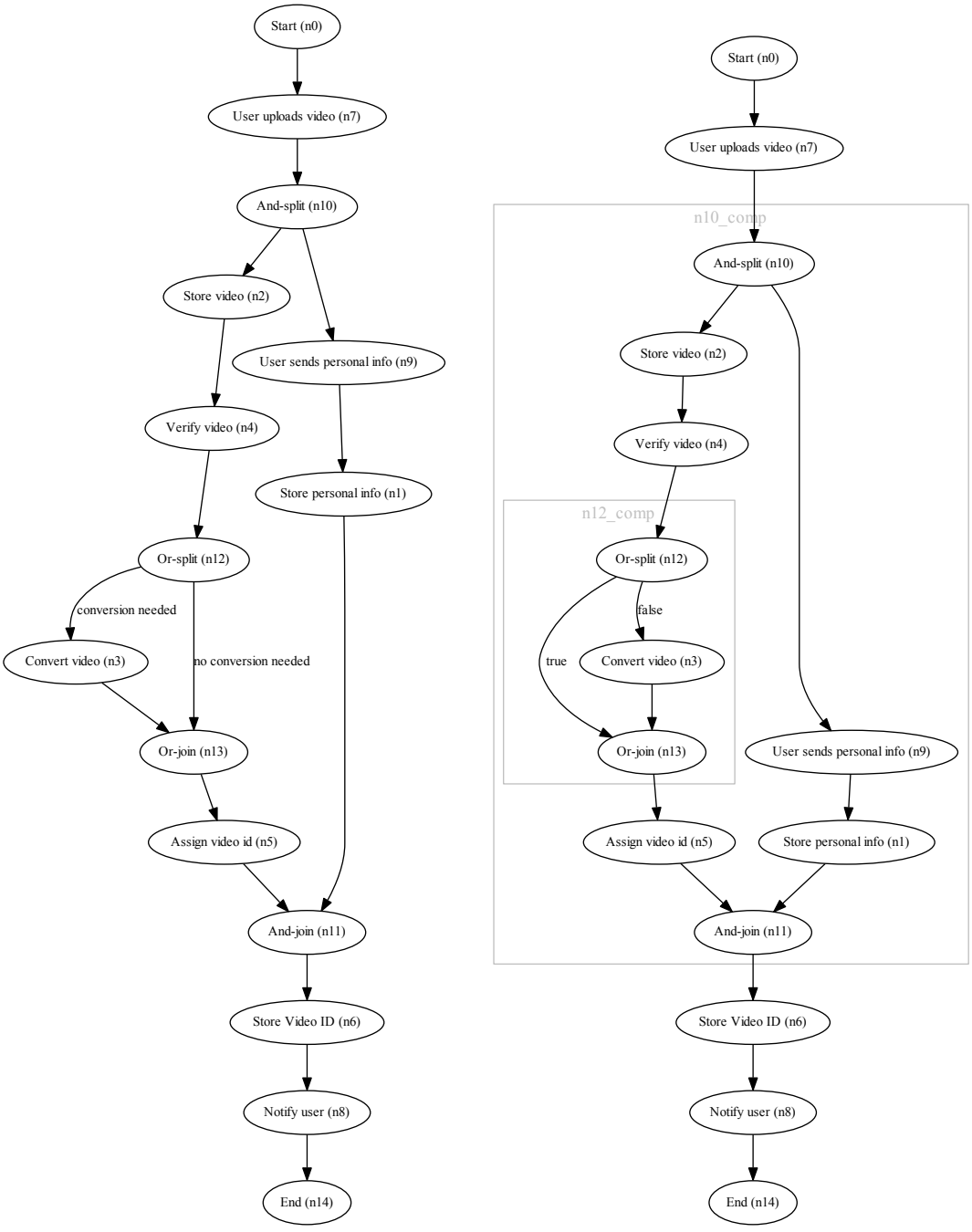
The first step of the lifting process is to export the business process to an XML representation. A fragment of the XML that was exported by BiZZdesigner is shown in Listing 9.1.

```
<?xml version="1.0" ?>
<orchestration>
  <nodes>
    <node id="n0" type="Trigger" name="Start" />
    <node id="n1" type="Action" name="Store personal info" />
    ...
  </nodes>
  <controlEdges>
    <controlEdge from="n11" to="n6" label="" />
    <controlEdge from="n12" to="n3" label="conversion needed" />
    ...
  </controlEdges>
  <dataItems>
    <dataItem name="Personal Information" />
    <dataItem name="Video" />
    ...
  </dataItems>
  <dataEdges>
    <dataEdge node="n1" dataItem="Personal Information" type="W" />
    <dataEdge node="n7" dataItem="Video" type="C" />
    ...
  </dataEdges>
  <distributionLocations>
    <distribution node="n0" location="OnPremise" />
    <distribution node="n2" location="Cloud" />
    ...
  </distributionLocations>
  <dataRestrictions>
    <restriction dataItem="Personal Information" location="OnPremise" />
  </dataRestrictions>
</orchestration>
```

Listing 9.1: XML fragment of the exported Amber process

9.3.2 Import

The exported XML file is imported by our Java application. A new instance of the intermediate model is created from the file. In addition to our Java transformations we also implemented a graphical export function to show the intermediate results during the transformations. Figure 9.3a shows the intermediate model that has been generatedfrom the imported XML file.



(a) After import (b) Replacement of composite constructs

Figure 9.3: Representation of the intermediate model

9.3.3 Replace constructs

Figure 9.3b shows the graphical representation of the intermediate model, in which the parallel and conditional nodes are captured within composite constructs.

9.3.4 Data dependency analysis

After the composite construct are created in the intermediate model, a data dependency analysis is performed. Figure 9.4 shows the intermediate model of the business process with the data dependencies between the items. We explain below a couple of data dependencies, shown in the Figure 9.4:

- Node n7 has a data dependency edge to itself, which means that the data item (in this case Video) is created during the execution of the activity.
- The activity Assign video id (n5) has a two data dependencies, both relating to the Video data item. The two incoming data dependencies mean, that both activities from which the data dependency edges originate are possible writers to the Video data item. Since the execution of the Convert Video (n3) activity is not guaranteed because it is in a conditional branch, activity n5 does not know if n3 has written to the item. Therefore, activity n2 is also a possible writer and a data dependency edge exists between n2 and n5.

9.4 Decomposition

The decomposition transformation can be started after the data dependencies have been determined. For each of the phases of the decomposition transformation, we will explain what happens with the process and show some of the intermediate results.

9.4.1 Identification

During the identification phase, the activities that need to be distributed in the cloud are marked. In addition, in each of the branches of a composite construct, a temporary node is added with the same distribution location as the composite construct.

9.4.2 Partitioning

In the partitioning phase, adjacent nodes marked with the same distribution location are placed together in a partition. This is shown in Figure 9.5. Each of the subgraphs

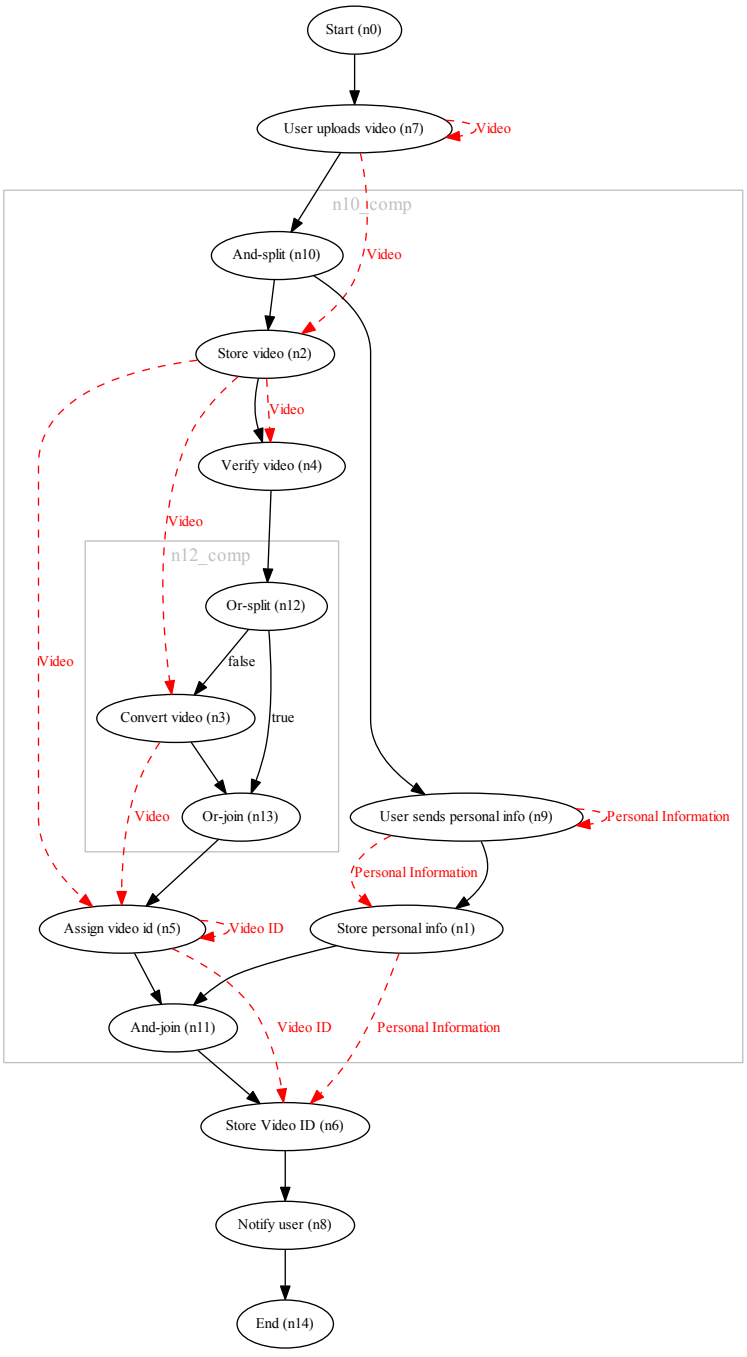


Figure 9.4: Data dependencies in the intermediate model

(branches) is treated as a separate process, therefore within a partition there might be multiple partitions within a composite construct. The shaded nodes in 9.5 represent the nodes that are marked for movement to the cloud, which were identified in the identification phase.

9.4.3 Creation of communicator nodes

During the next phase, communicators are created between partitions. Consider Partition1 and Partition2 in Figure 9.5. Partition1 is allocated on-premise and Partition2 is marked for movement to the cloud. Partition1 is extended with invocation nodes and Partition2 is extended with a receive-node at the beginning of the partition and a reply-node at the end of the partition.

9.4.4 Choreography creation

After the communicators are created, the separate processes are collected and the temporary nodes that were added in the identification phase are removed. The result that is obtained after this phase is shown in Figure 9.6.

9.4.5 Data restriction verification

The decomposition transformations are finished now and data restriction verification is needed to verify that no data restrictions were violated during the transformations. The verification algorithm collects the data items that were violated, and selects the activities that violate these data items. The information obtained in this phase is used during the grounding transformation. In this example, no data restrictions are violated.

9.5 Grounding

The next step in the transformation chain is the grounding transformation, in which the intermediate model is transformed back to an Amber model.

9.5.1 Export

In order to allow BiZZdesigner to display the result of the decomposition transformation, the intermediate model is transformed into a format that can be imported by BiZZde-

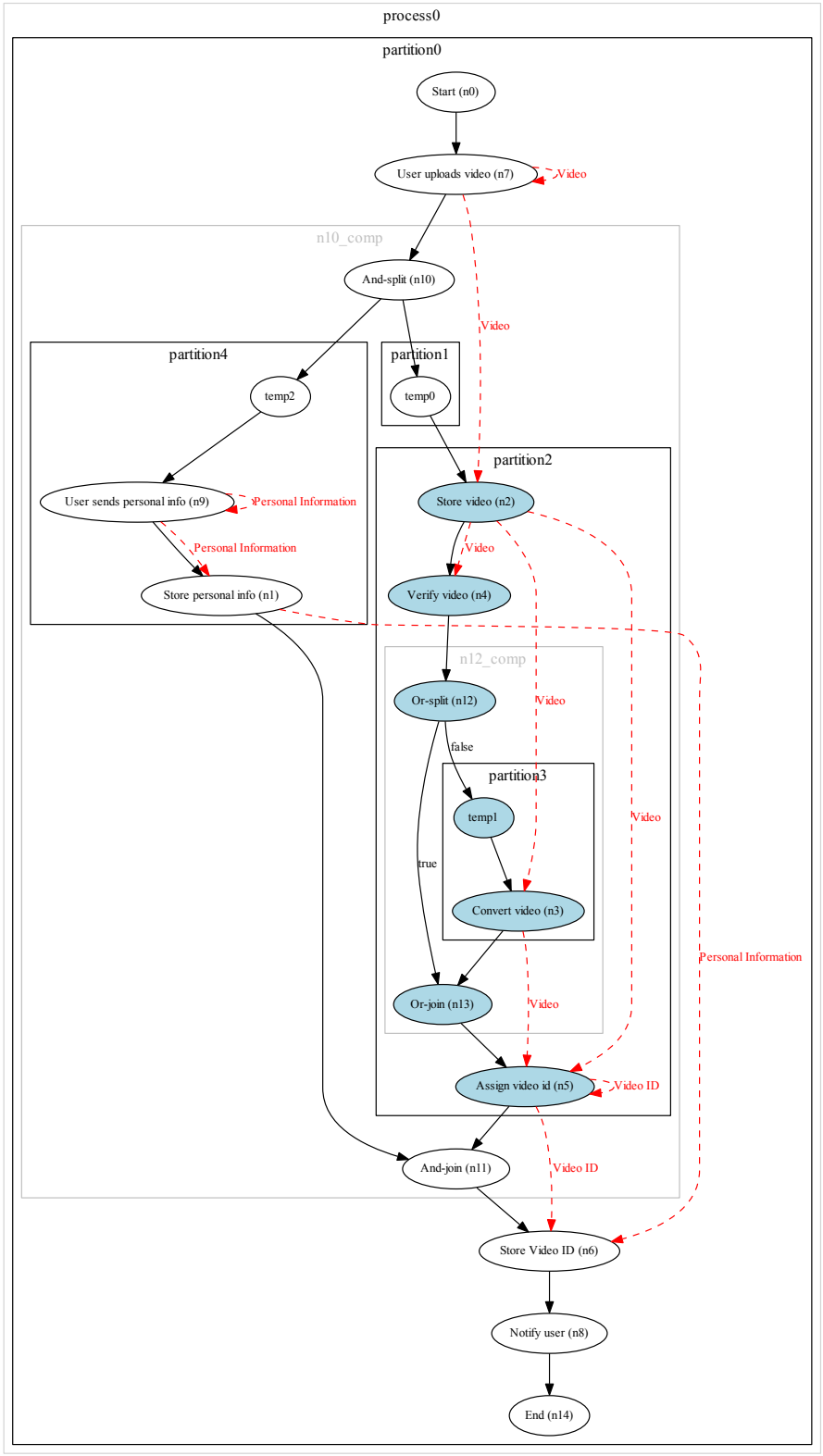


Figure 9.5: Intermediate model after the partitioning phase

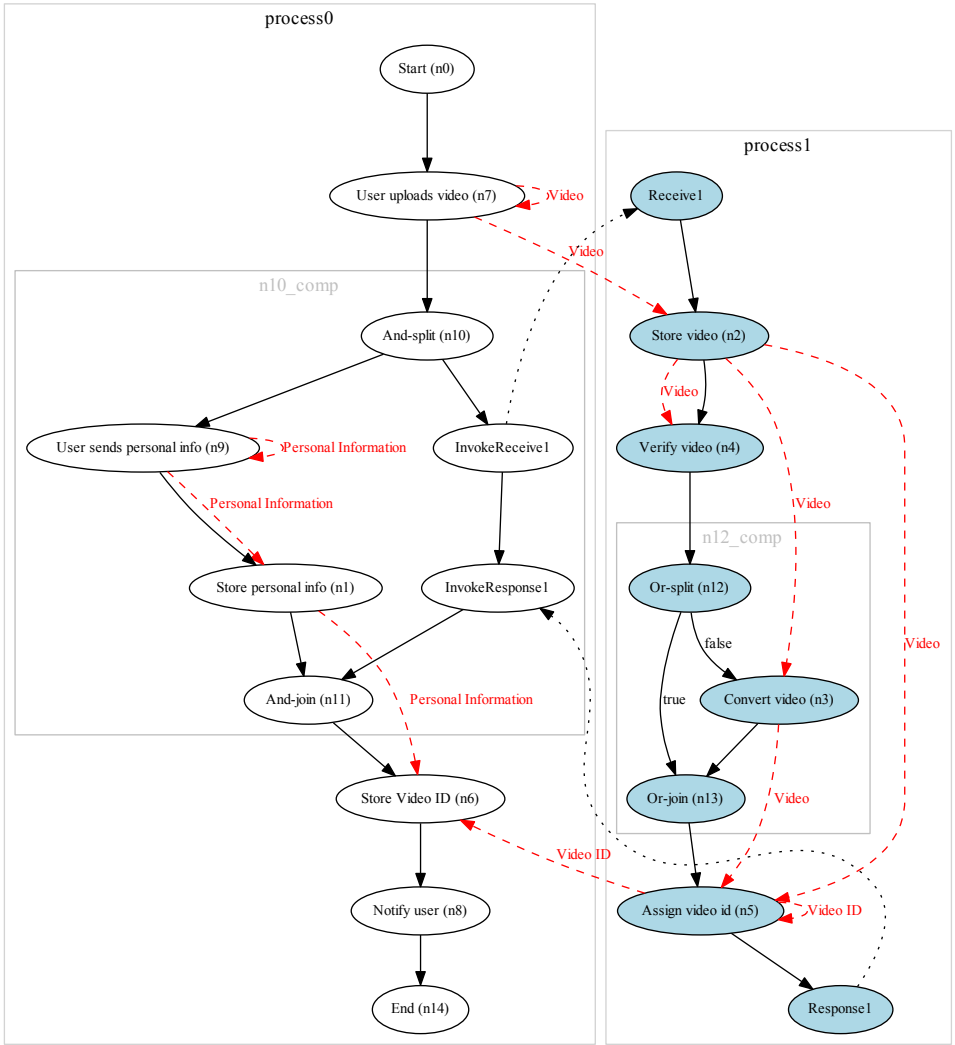


Figure 9.6: Intermediate model after the choreography creation phase

signer. Part of the XML output that was generated by the grounding transformation is shown in Listing 9.2.

```
<?xml version="1.0" ?>
<choreography>
  <dataItems>
    <dataItem name="Video" />
    <dataItem name="Personal Information" distributionLocation="OnPremise
        " />
    ...
  </dataItems>
  <orchestrations>
    <orchestration name="process0" distributionLocation="OnPremise">
      <nodes>
        <node id="n0" name="Start" type="Trigger" />
        ...
      </nodes>
      <edges>
        <edge from="n0" to="n7" type="Control" label="" />
        ...
      </edges>
    </orchestration>
    <orchestration name="process1" distributionLocation="Cloud">
      <nodes>
        <node id="Receive1" name="Receive1" type="Interaction" />
        ...
      </nodes>
      <edges>
        <edge from="Receive1" to="n2" type="Control" label="" />
        ...
      </edges>
    </orchestration>
  </orchestrations>
  <interactionEdges>
    <edge from="InvokeReceive1" to="Receive1" type="Communication" />
    ...
  </interactionEdges>
  <dataEdges>
    <edge from="n2" to="Video" type="Data" relationType="W" />
    ...
  </dataEdges>
</choreography>
```

Listing 9.2: XML fragment of the created choreography

9.5.2 Import

During the import phase, the XML format is converted into a new behavioral model in BiZZdesigner. The resulting process is shown in Figure 9.7. The result consists of two collaborating processes. The first process is meant for deployment on-premise and invokes the second process, which is meant for deployment in the cloud.

9.6 Example of data restriction violation

The example we have presented did not violate any data restrictions. However, we show in the sequal what happens when a data violation is introduced. By updating the business

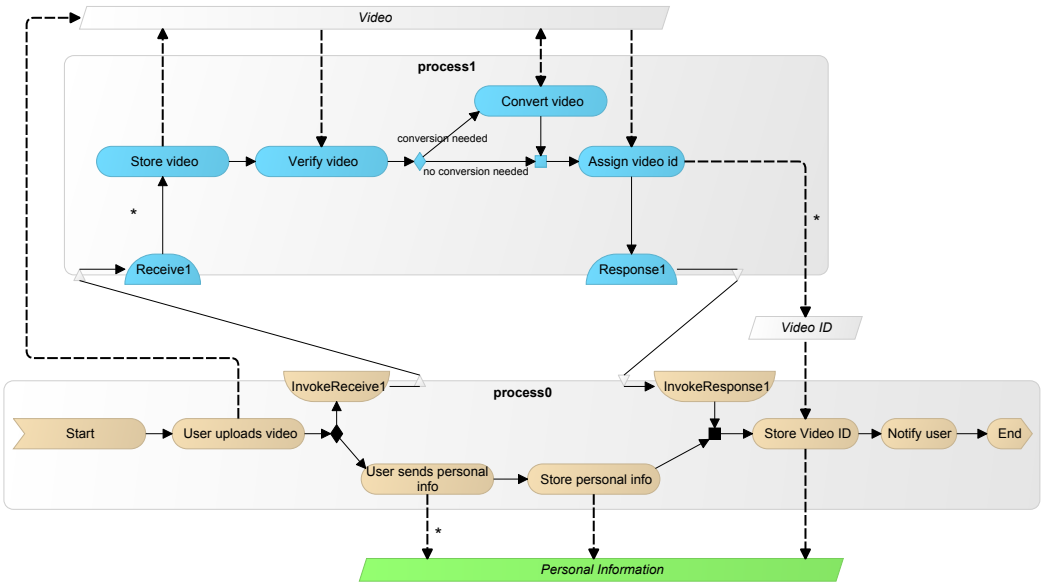


Figure 9.7: Business process with marked activities and data restrictions

process and marking the Store Video ID activity for movement to the cloud, as shown in Figure 9.8 a data restriction will be violated during the transformations.

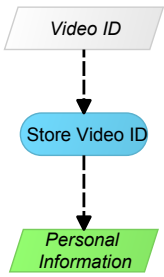


Figure 9.8: Business process with marked activities and data restrictions

Figure 9.9 shows the result after the applying the transformation chain on the business process. For violated data items and activities that violate a data restriction, a special flag is set. By setting this flag, the items will be colored red in the resulting Figure. The data item Personal Information is colored red, which indicates that the data restriction on the data item is violated. The red colored nodes (Receive0 and Store Video ID) are the nodes that violate the data restriction.

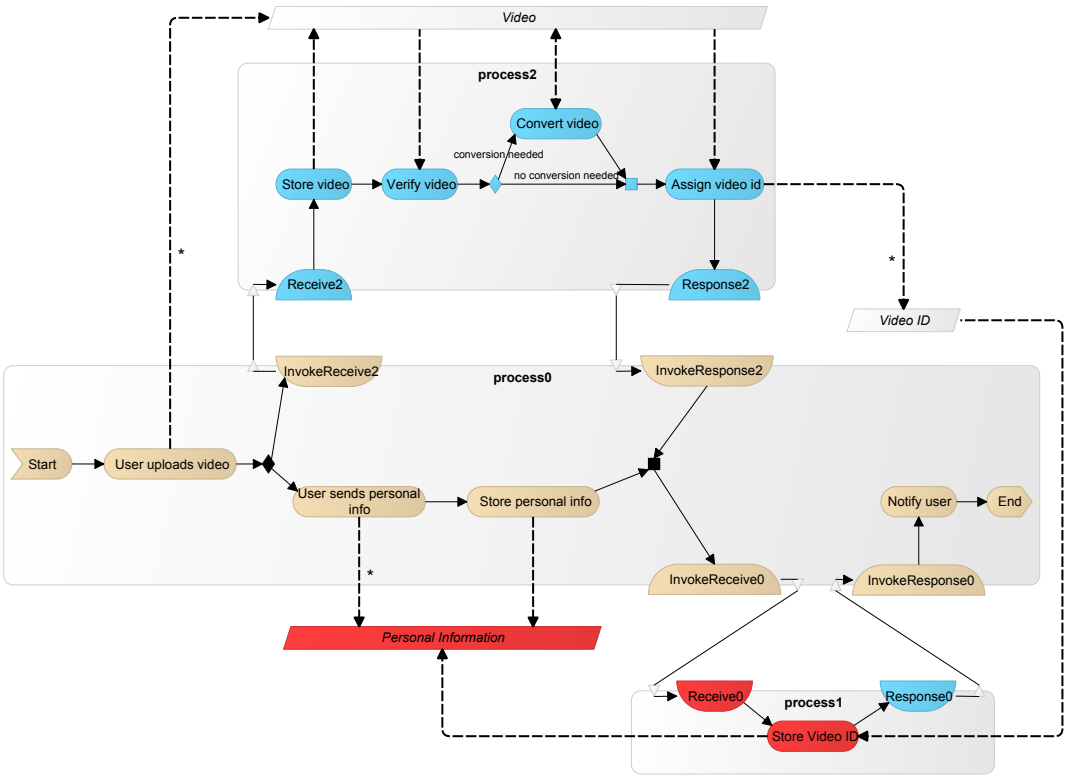


Figure 9.9: Business process with marked activities and data restrictions

9.7 Conclusion

This case study demonstrates that the transformations can be performed automatically. The Java implementation has been extended with a function to export intermediate results as images. The initial process was created by hand and the marking of activities and data item was also performed by hand. The layout of the resulting model was corrected by hand, since BiZZdesigner has no automatic layout functionality.

10. Conclusions

This chapter provides the conclusions of our work. We describe the general conclusions, answer the research questions and identify possible future work.

10.1 General Conclusions

In this thesis, we explained the design and implementation of a decomposition framework for decomposing monolithic business processes into multiple processes that can be executed in the cloud or on-premise. The decomposition is driven by a distribution list, in which the activities of the original business process are marked with their desired distribution locations, and data restrictions can be added, to ensure that data items stay within a certain location (on premise or in the cloud).

We explained the BPM lifecycle and cloud computing and introduced an already existing solution, in which an architecture was built for combining traditional BPM with cloud-based BPM. We extended this work by identifying a new distribution pattern, in which process engines are placed both on-premise and in the cloud.

A transformation chain was defined for our decomposition framework. The decision was made to use an intermediate model for defining the decomposition transformation. This intermediate model is a semantic model in which the main concepts of a business process are captured. The decomposition transformation was designed for working on the intermediate model. By performing the operations on the intermediate model, the decomposition solution is business process language-independent and is suitable for both processes defined in the design and implementation phase of the BPM lifecycle. In order to work with existing business process languages, transformations are needed for converting an existing business process language into the intermediate model and back, the so called lifting and ground transformation, respectively.

An analysis was performed to identify the decomposition rules that should be supported. From these rules, a selection was made for the implementation of the transformation. The algorithm that was used for the decomposition transformation was first implemented using graph transformations. After that, the algorithm was implemented in Java. We also built a data verification algorithm to verify if data restrictions were violated as a result of the decomposition transformation.

We selected Amber [24] as the business process language, and developed the lifting and grounding transformations for this language. Algorithms were designed for replacing conditional, parallel and loop nodes by block structured elements, and a data dependency analysis algorithm was designed for discovering data dependencies between activities.

We tested the decomposition framework on multiple examples. In this thesis, we used the talent show audition case study, to show how the framework can be applied to real-life business processes.

10.2 Answers to the research questions

The research questions that we defined for this thesis have been answered:

RQ1. Which approaches are available for decomposing a business process?
In Chapter 3.2, we identified related work on the decomposition of processes. Most of the discovered work focuses on the BPEL [6] language. Our work focuses mainly on the design level of the BPM lifecycle, since the decision for deployment locations of data and activities is not only a choice at the implementation level. At the design level, issues such as security of data and governance rules already influence if activities can be placed in the cloud or on-premise. The approaches that were used in the related work focused on language specific issues, whereas general information about the decomposition of processes was not available. Eventually, we based our intermediate model on models used in related work, and defined our own decomposition rules.

RQ2. Which transformation rules can be applied to constructs within a business process?
In Chapter 5 we analyzed possible transformation rules for each of the constructs that are available in our intermediate language. We made a selection from these solutions for our decomposition transformation and implemented them.

RQ3. How to deal with data restrictions when decomposing a business process?
In order to identify the consequences for data restrictions when decomposing a business process, data dependencies were added to the intermediate model for capturing the data relations between nodes in the process. The data dependencies were defined by performing a data analysis on the original process. After the execution of the decomposition transformation, a data verification algorithm is used for validating that no data restrictions were violated during the transformation.

RQ4. How to verify the correctness of the decomposition solution?
We analyzed the main transformation rules for each of the types of nodes that are available in the intermediate model. The transformation rules are recursively applied to graphs and graph fragments within composite constructs. By taking the result of the decomposition transformation and replacing the communicators by control edges, the original process can be obtained. This means that during the transformation, no information is lost from the process and the behavior of the process

has not changed. In future work, formal verification can be used for proving the correctness of the decomposition transformation.

10.3 Future Work

During the implementation of our work, we identified several research opportunities for future work:

Implementation

In our work, we focused on the design level of the BPM lifecycle and on the decomposition rules for business processes. Deployment of the processes on-premise and in the cloud, however, has not been tested, since our base language is not executable. By choosing for an executable business process language as base language for the lifting and grounding transformations, the deployment of the newly created processes can be tested and the behavior of the newly created processes can be compared with the original process at runtime.

Extend intermediate model

Before we defined our intermediate model, we selected requirements for this model, namely a subset of the workflow patterns in [23]. In future work, the intermediate model can be extended to support more workflow patterns. In addition, the model can be extended to model exceptional behavior. This extension is needed when dealing with process languages such as WS-BPEL [6] or BPMN2.0 [5, 25].

Implement additional decomposition rules

In our work, we performed an analysis of possible decomposition rules. We identified solutions and made design decisions on these solutions. The decomposition transformation can be extended with more complete transformations, such as the support of composite constructs in which the start and end nodes have different distribution locations.

Extend distribution locations

Our work considers two possible distribution locations: the cloud an on-premise. In future work, the number of distribution locations could be extended. This gives organizations the opportunity to use multiple cloud vendors and define multiple on-premise locations, for example for distributing parts of a business process to different departments within an organization, or distributing parts of a process to other organizations.

Calculation and recommendations

In our work, we did not focus on the actual costs of the original process and the

created processes. In future work, a calculation framework can be designed and implemented to take costs into account. In addition, the framework should give recommendations concerning which activities or data should be placed at which location. Formulas for calculating the actual costs of distributing activities and data in the cloud were defined in [4]. These formulas can be extended and used for the calculation framework. Research is needed to identify all the factors that influence the costs of using BPM on-premise or in the cloud.

A. Graph transformation

During this project we used Groove [30] for implementing the graph transformations. The graph transformations were only used to test our transformation strategy and for more formally defining our transformation steps. In the remainder of this chapter we introduce the concept of graph transformations, show the type graph that was used and the rules that were created for our decomposition transformation.

A.1 Introduction

Groove [30] is the graph transformation tool that was used for testing and implementing our decomposition transformation. We will give a very brief overview of the concepts that can be expressed with the language. More information about graph transformations can be found in [30].

A graph transformation consists of an initial model and a set of rules that can be applied to the graph. Each rule matches a pattern in the source graph and makes changes to the graph. The rules are composed graphically in Groove. A type graph is used for capturing the possible nodes and relations that may occur in the graph.

A graph transformation rule may consists of:

- **Matching elements**
  Elements that should be matched in the graph. These elements are displayed in Groove in gray. Relations between elements that are matched are displayed by black lines.

- **Removing elements**
  Elements that are marked for deletion are colored blue. The transformation engine will match the elements in the graph and eventually delete them when executing the rule.

- **Creating elements**
  The green elements in the transformation rule are the elements that will be created by the transformation engine, when executing the rule.

- **Embargo**
  An embargo can be used to define elements or relations that should not be present in the graph.

A.2 Type Graph

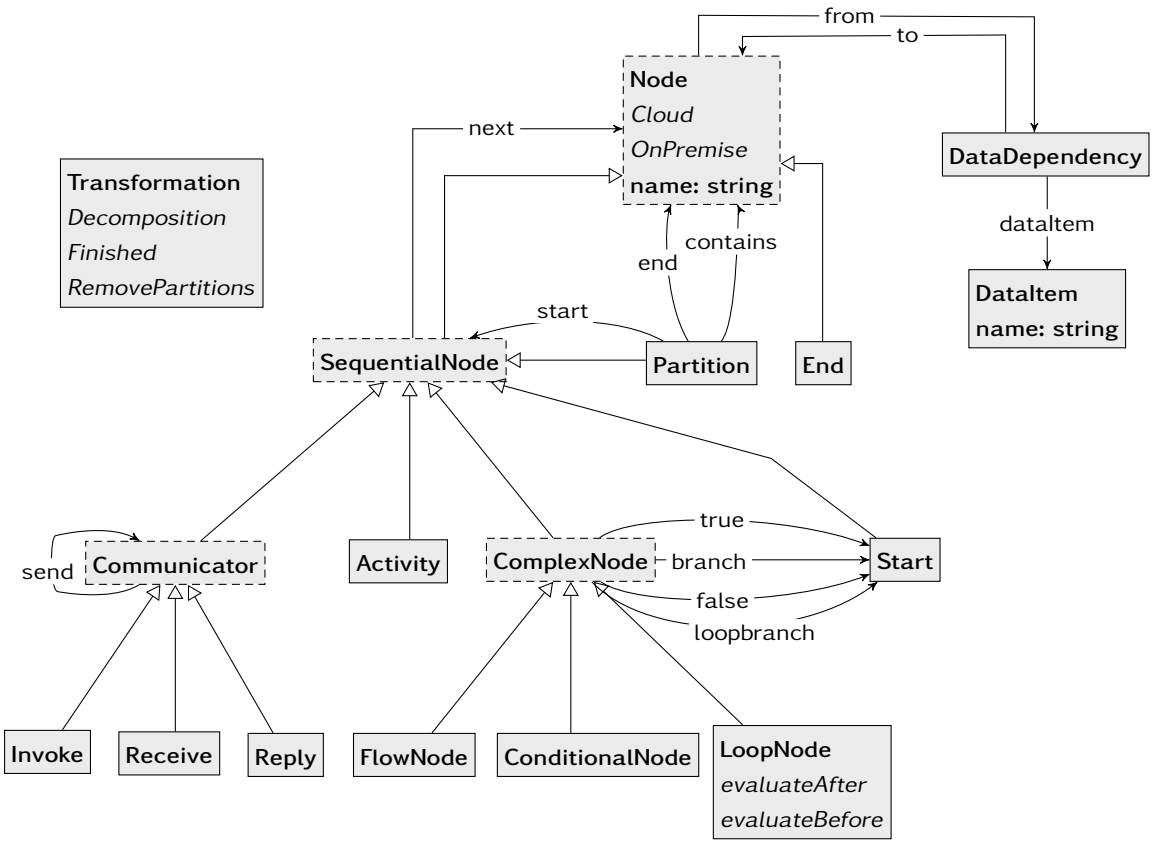


Figure A.1: Type Graph of the intermediate model in Groove

Figure A.1 shows the type graph that is used for the graph transformation. In this section we give an overview of the types that are defined in the typegraph and the relation between the elements.

A graph model consists of nodes which are connected to each other by edges. A process does always start with a "Start" node and ends with an "End" node. The following nodes are available in the type graph:

Node
The "Node" type is an abstract type from which all the possible node types inherit. The type is provided with two possible flags (OnPremise and Cloud) for marking a node with its distribution location. In addition, a name attribute is added to the type, to be able to attach an identifier to a node.

SequentialNode
"SequentialNode" is an abstract subtype of "Node", which is used as parent node for all

types that have a direct successor node. As a consequence, inheritance from this type has an "next" edge, which points to the successor node. The only node type that does not inherit from the "SequentialNode" type is the "End" node, since it does not have a successor.

- **Start**
  "Start" nodes are used for marking the beginning of a process. Not only the main process starts with this node, but also branches of composite constructs start with a "Start" node.

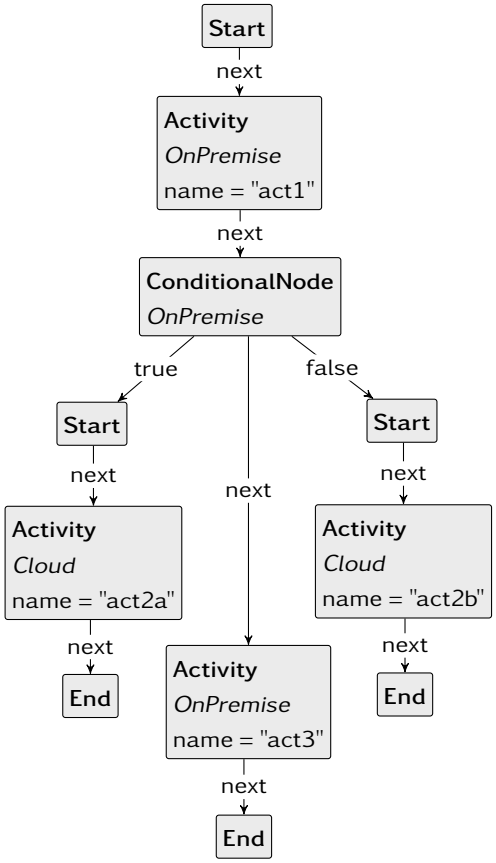
- **Activity**
  Activities within a process are modeled by using "Activity" nodes.

- **Communicator**
  Three types of communicators are available in the type graph: "Invoke", "Receive" and "Reply". The nodes correspond to the nodes defined in the intermediate model. A communicator has an incoming and/or outgoing send node, which defines communication between two communicators. The intuition of the "Invoke" node is that, when the node only has an outgoing send edge, the process is asynchronous and continues after sending a request to another process. When the node has an outgoing send edge and an incoming send edge, the node first invokes the other process and waits until the "reply" node replies to the "invoke" node.

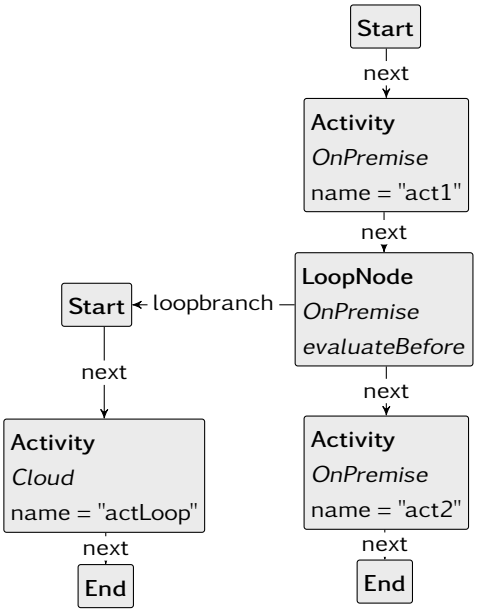
- **Partition**
  A "Partition" node is used for grouping sequences of nodes with the same distribution location. The node has outgoing "contains" edges to all the nodes that are contained by the partition. The start and end node of a partition are marked with an additional start and end edge.

- **Complex nodes**
  "ComplexNode" Is the super type for nodes with multiple branches. The outgoing edges defined for this type are used in combination with specific sub types of the node. The choice has been made to define these edges on the abstract "ComplexNode" type, even when they are not valid for certain sub constructs, to avoid that additional transformation rules need to be defined for each of the specific types. The following node types inherit from the "ComplexNode" type:

  - **FlowNode**
    A "FlowNode" is used for defining parallel behavior. The node has two or more branch edges, which point to subprocesses that are executed in parallel. The outgoing next edge of the node corresponds to the path which is taken after execution of both branches is finished. Instead of using a join node, the intuition



(a) Situation with a conditional construct



(b) Situation with a loop

Figure A.2: Type Graph of the intermediate model in Groove

of the construct is that the "FlowNode" waits until both branches are finished, before the next edge is followed.

– **ConditionalNode**
The "ConditionalNode" is used for defining conditional behavior. The node has two outgoing edges, labeled with either true or false, which correspond to the evaluated condition in the "ConditionalNode". Likewise as with the "FlowNode", the "ConditionalNode" defines a next edge, which points to the path that is taken after the branch is executed. An example of usage of conditional constructs in our graph model is shown in Figure A.2a.

– **LoopNode**
The "LoopNode" is used for defining loops in the graph. The node has an outgoing loopbranch edge, which points to the subprocess which is executed in the loop. The node can be marked with a evaluateAfter or evaluateBefore flag, which defines if the condition should be evaluated at the beginning of the loop

or at the end of the loop. The next edge points to the node that are executed when the loop condition does not longer hold. An example of a graph with a loop is shown in Figure A.2b

DataDependency
A "DataDependency" can be used for modeling data dependencies between nodes. A data dependency between two nodes is created by creating a "DataDependency" object which has an incoming from edge originating from the node in which a certain data item is defined or changed. The outgoing to edge points to the node which uses the data item. The data dependency also points to a "DataItem" object, which contains the data item that is used.

Transformation
The "Transformation" type is used during the graph transformation for marking the phase of the transformation. The graph transformation consists of two phases: the decomposition phase and the partition removal phase. The rules of the transformation are prioritized to define the execution sequence of the rules. Some rules defined in the phases might conflict. For example, in the decomposition phase the graph transformation tries to define partitions for adjacent nodes with the same distribution location. In the removal phase however, these partitions are merged and removed again. By removing a partition, the earlier defined rules in the decomposition phase are applicable again, which might cause an infinite loop in the graph transformation. A solution would be to split the phases into two separate transformations, but Groove has no support for this. Instead, we decided to introduce a specific node for marking the current phase of the transformation. A "Transformation" type is created a the beginning of the overall transformation, which defines the state in which the graph transformation is working. During the decomposition phase, only decomposition rules are applicable and during the partition removal phase, the decomposition rules are disabled.

A.3 Transformation Rules

A.3.1 Phases and priorities

The graph transformation consists of two phases. At first, the decomposition phase is used in which nodes are grouped in partitions and communicators between partitions are created. The second phase is the partition removal phase, in which all the partitions are removed again, so that only the processes and communication edges between the processes remain.

Table A.1 shows an overview of all the rules that are used and the priority of the rules.

Rules with higher priority take precedence over rules with lower priority. The execution sequence of rules with the same priority is at random.

Phase	Rule name	Priority
Decomposition	startDecompositionPhase	8
Decomposition	replaceCompositeNodes	7
Decomposition	mergePartitions	7
Decomposition	addActivityToPartition	6
Decomposition	createStartPartition	5
Decomposition	createFollowUpPartition	5
Decomposition	createCommunicatorBranch	4
Decomposition	createCommunicator	4
Partition Removal	startPartitionRemovalPhase	3
Partition Removal	removePartition	2
Partition Removal	removeFirstPartition	1
Partition Removal	finishTransformation	0

Table A.1: Transformation rules with their priority

In the remainder of this section we will explain each of the transformation rules in more detail.

A.3.2 Rules

Rule startDecompositionPhase (Priority 8)

This rule starts the decomposition phase. The constraint in the rule specifies that their should not be any transformation currently going on and their should not be communicators, since communicators indicate that the process is already a collaboration with multiple processes. A transformation node is created and the decomposition phase is selected as next phase in the transformation. Figure A.3 shows the graphical representation of the rule.

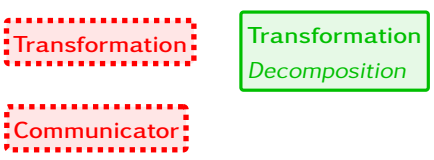


Figure A.3: Definition of startDecompositionPhase in Groove

Rule replaceCompositeNodes (Priority 7)

When dealing with a complex node, the start nodes of the branches should be marked with the same distribution location as the complex node itself. This is necessary in a later stadium of the transformation, when partitions are created. The rule matches the distribution location, which is stored in the distrLoc variable. When the start node in a branch is not yet marked with the distribution location, the distribution location is set to the node. The graphical representation of the rule is shown in Figure A.4.

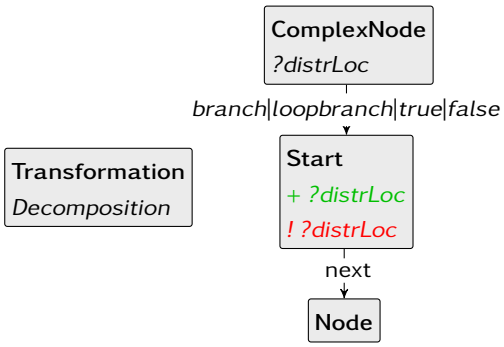


Figure A.4: Definition of replaceCompositeNodes in Groove

Rule mergePartitions (Priority 7)

The mergePartitions rule is performed when two sequential partitions have the same distribution location. The second partition will be removed and all the nodes from the second partition will be added to the first partition. In addition, a "next" edge will be created from the "end" node of the first partition to the "start" node of the second partition. A new "end" edge will be created from the first partition to the last node of the partition that will be removed. The "next" edge of the first partition will point to the succeeder of the partition that will be removed. The graphical representation of the rule is shown in Figure A.5.

Rule addActivityToPartition (Priority 6)

When a Partition is followed by SequentialNode with the same distribution location as the Partition, the SequentialNode should be added to the partition. A new next edge will be created between the last node in the Partition and the SequentialNode. The partition needs a new end edge pointing to the SequentialNode and the partitions next edge points now to the succeeder of the SequentialNode. The graphical representation of the rule is shown in Figure A.6.

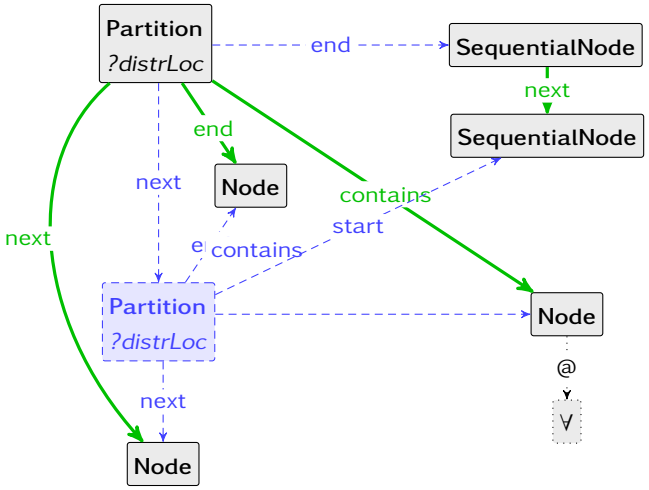


Figure A.5: Definition of mergePartitions in Groove

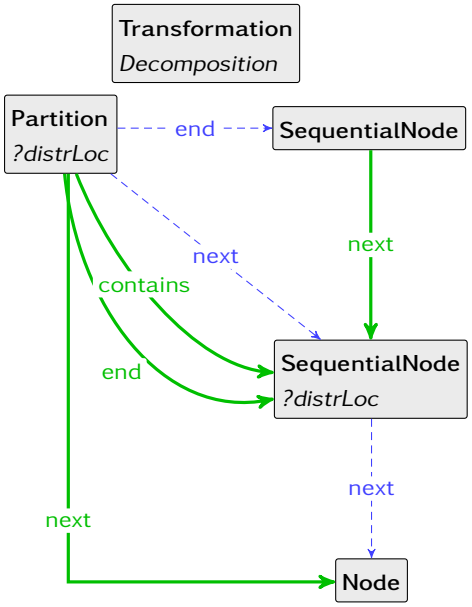


Figure A.6: Definition of addActivityToPartition in Groove

Rule createStartPartition (Priority 5)

The createStartPartition rule creates a partition for the first node that is defined after a Start node. The rule ensures that the node is not yet placed in a partition. A new partition is created and the edges “start”, “contains” and “end” are created. The “next” edges from the Start node to the SequentialNode and from the SequentialNode to the following Node (n2) are removed and new “next” edges are placed from the Start node to the Partition and from the Partition to the following. The graphical representation of the rule is shown in Figure A.7.

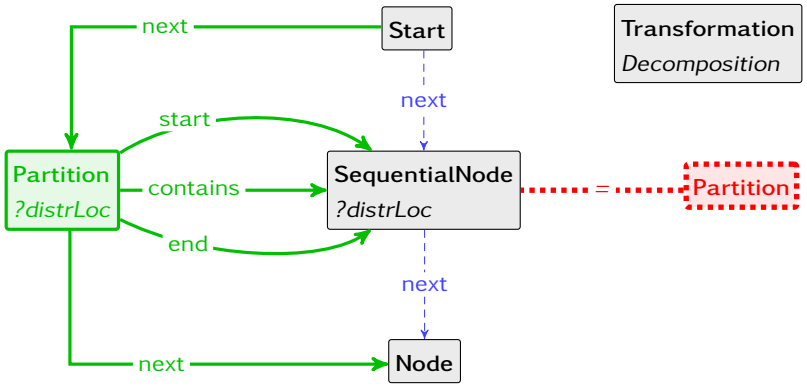


Figure A.7: Definition of createStartPartition in Groove

Rule createFollowUpPartition (Priority 5)

When a Partition is followed by a SequentialNode, with a different distribution location as the Partition and the SequentialNode is not yet placed in a Partition, then a new Partition should be created for the node. The new Partition will be marked with the distribution location of the SequentialNode and will point to the SequentialNode with the "start", "contains" and "end" edges. A new "next" edge will be created between the partitions. The graphical representation of the rule is shown in Figure A.8.

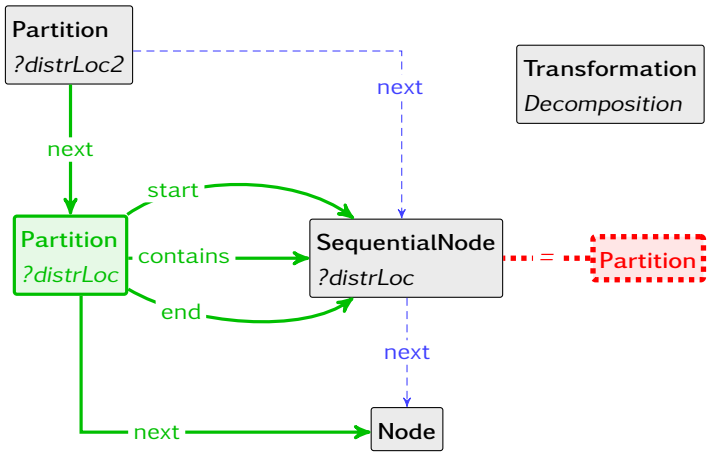


Figure A.8: Definition of createFollowUpPartition in Groove

Rule createCommunicatorBranch (Priority 4)

The createCommunicatorBranch rule is used when the first partition (p1) inside a branch has a different distribution location as the complex construct from which the branch originates. In this situation a new partition (p2) will be added to the branch in which an Invoke

node is created for invoking the partition p1. Partition p1 will be extended with a Receive and a Reply node. The graphical representation of the rule is shown in Figure A.9.

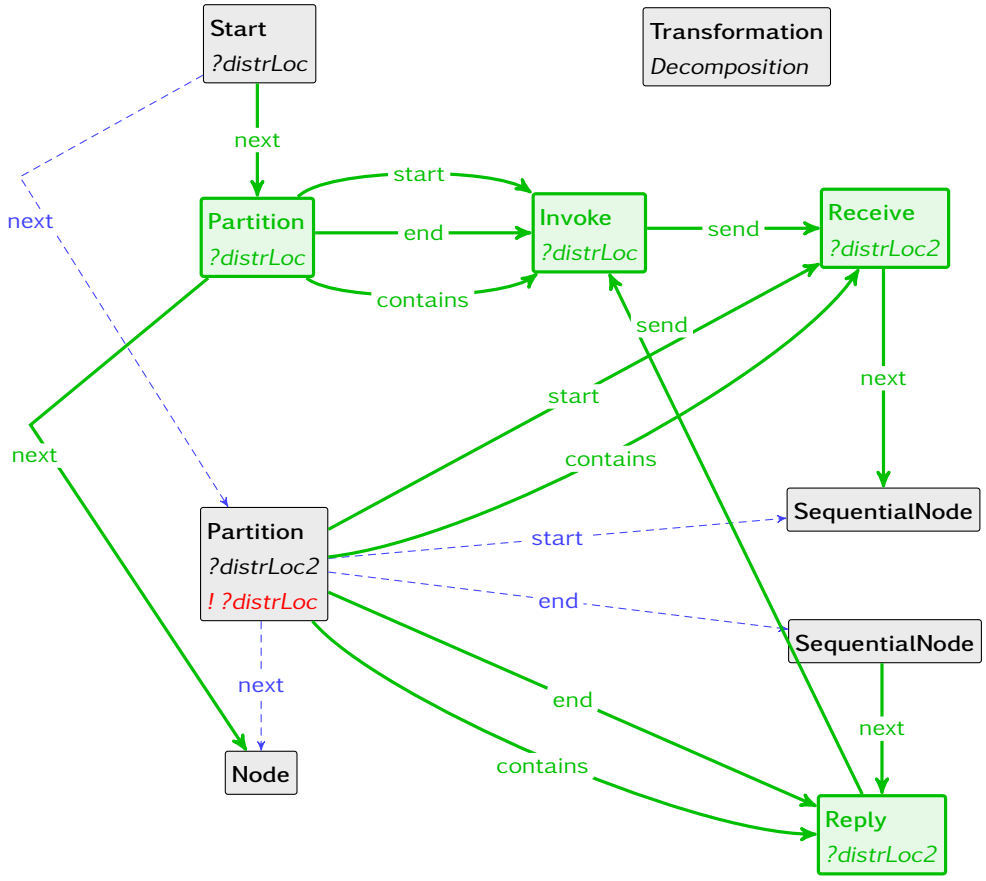


Figure A.9: Definition of createCommunicatorBranch in Groove

Rule createCommunicator (Priority 4)

The createCommunicator rule creates communication nodes between two partitions with a different distribution location. The first partition will be extended with an Invoke node, which synchronously invokes the second partition. The second partition will be extended with a Receive and a Reply node. The second partition will be removed from the original process, since it is now an individual process. The graphical representation of the rule is shown in Figure A.10.

Rule startPartitionRemovalPhase (Priority 3)

The rule is used for changing from the decomposition to the remove partitions phase. The flag of the Transformation node is changed by the rule. The graphical representation of

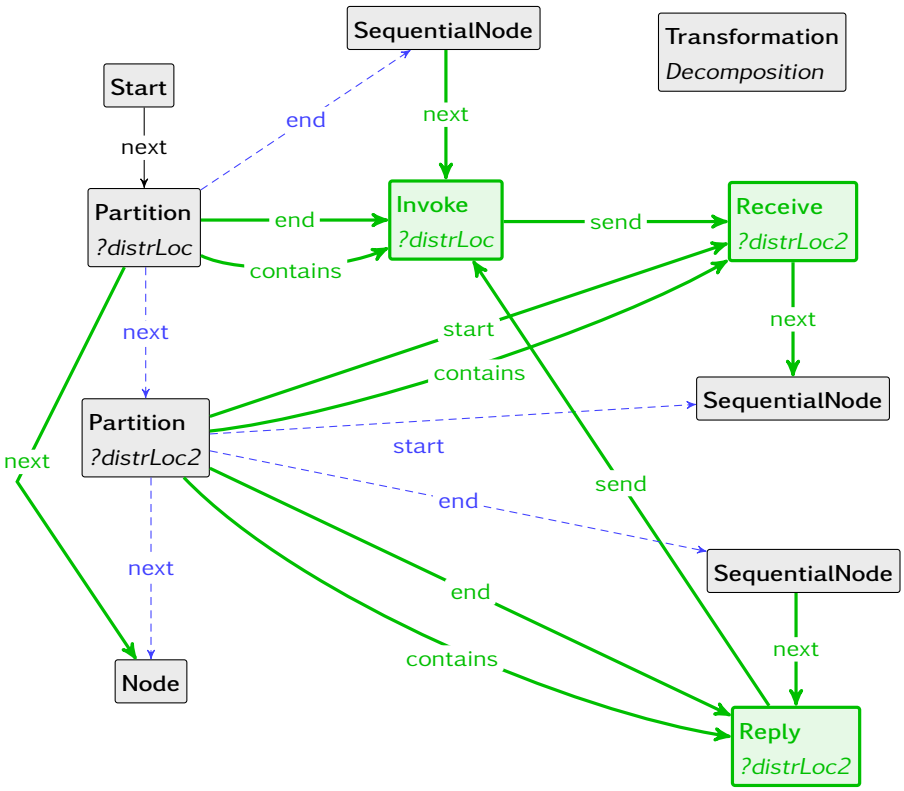


Figure A.10: Definition of createCommunicator in Groove

the rule is shown in Figure A.11.

Figure A.11: Definition of startPartitionRemovalPhase in Groove

Rule removePartition (Priority 2)

The removePartition rule removes a partition and connects the contents of the partition with the node before the partition and the node after the partition. The graphical representation of the rule is shown in Figure A.12.

Rule removeFirstPartition (Priority 1)

This rule matches Partitions that have no incoming and outgoing next nodes. The graphical representation of the rule is shown in Figure A.13.

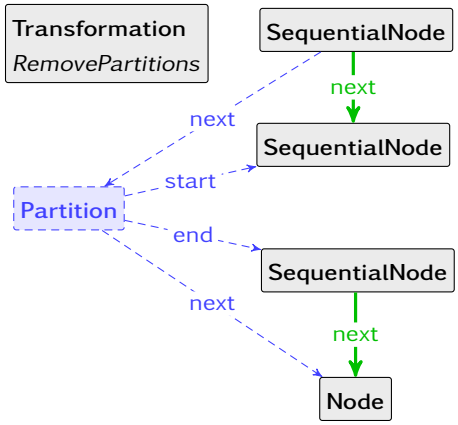


Figure A.12: Definition of removePartition in Groove

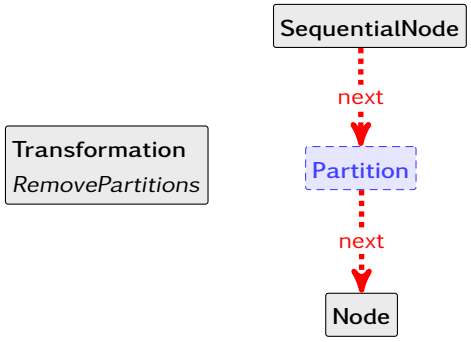


Figure A.13: Definition of removeFirstPartition in Groove

Rule finishTransformation (Priority 0)

When all the previous rules are applied, the finishTransformation rule is used to update the Transformation node and change the phase from the remove partitions phase to finished. The graphical representation of the rule is shown in Figure A.14.

Figure A.14: Definition of finishTransformation in Groove

A.4 Example

In Figure A.15, an example is shown of a process defined using our type graph. By applying the introduced rules, the graph in Figure A.16 is obtained.

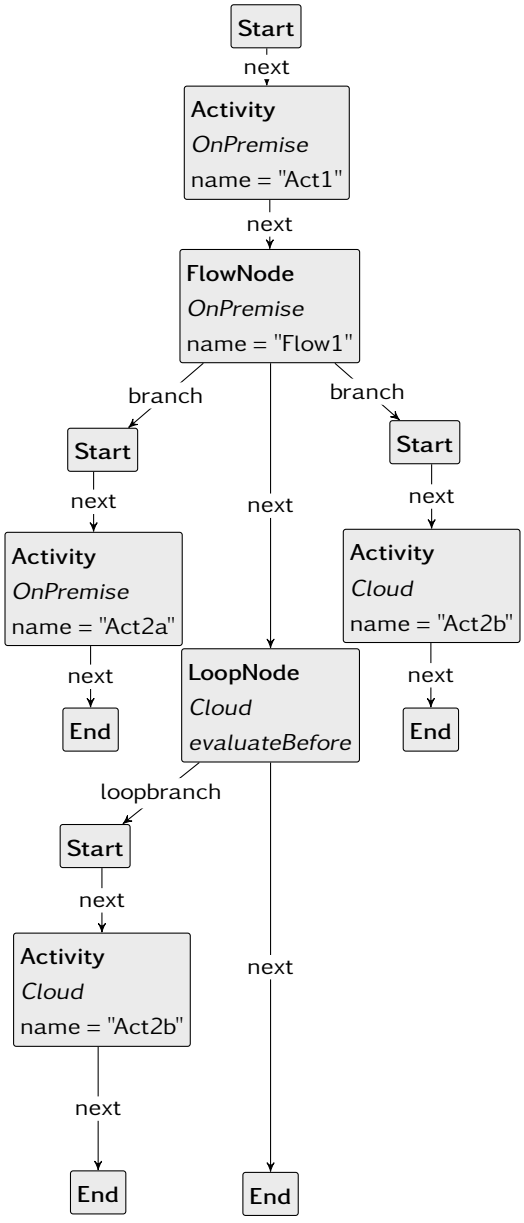


Figure A.15: Example of a process in Groove

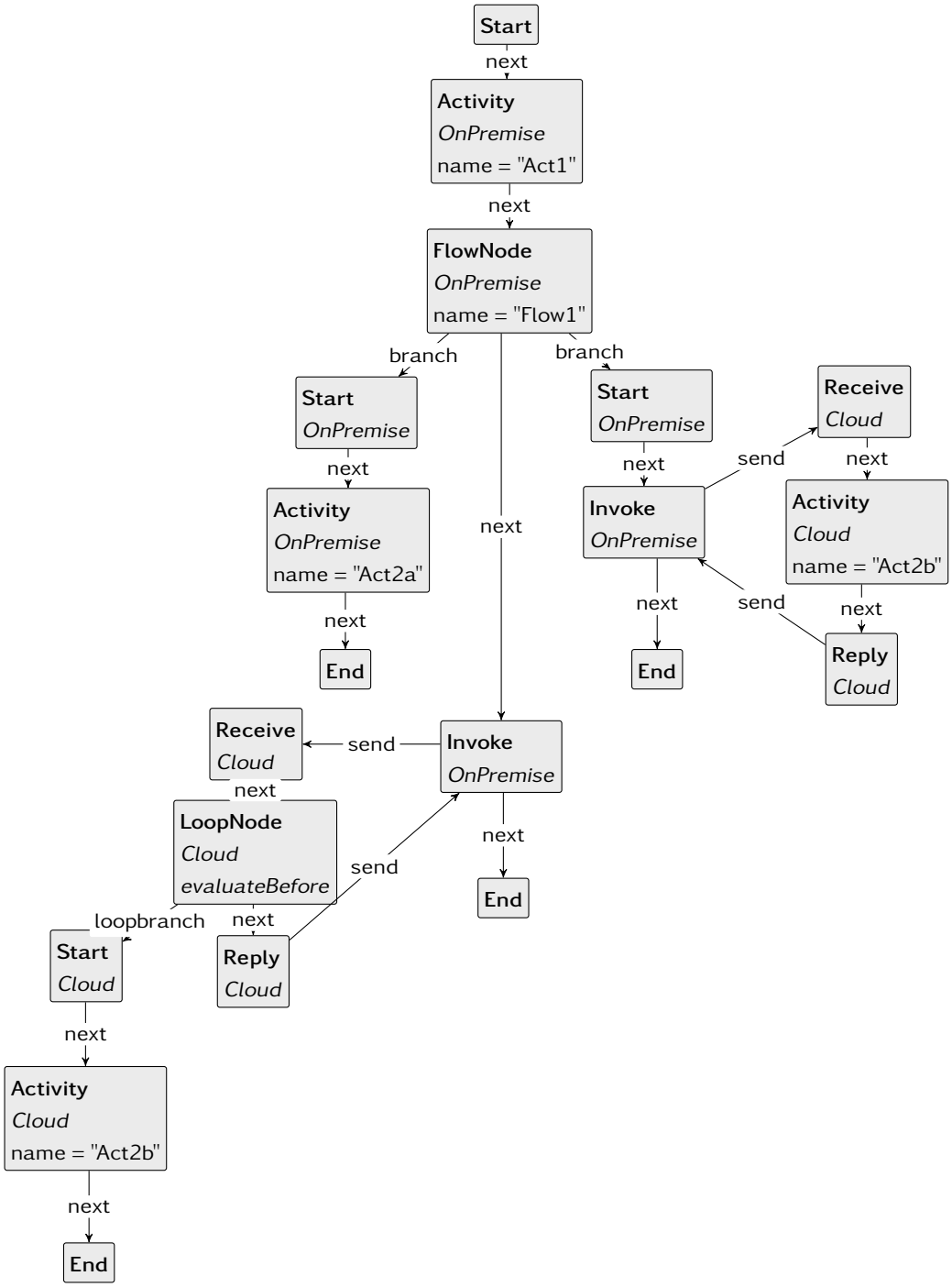


Figure A.16: Result after the transformation in Groove

Bibliography

[1] M. Weske, *Business Process Management: Concepts, Languages, Architectures*. Springer, 2007.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[3] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *National Institute of Standards and Technology*, vol. 53, no. 6, p. 50, 2009.

[4] Y.-B. Han, J.-Y. Sun, G.-L. Wang, and H.-F. Li, "A cloud-based bpm architecture with user-end distribution of non-compute-intensive activities and sensitive data," *J. Comput. Sci. Technol.*, vol. 25, no. 6, pp. 1157–1167, 2010.

[5] O. M. Group, "Business Process Model and Notation (BPMN) Version 2.0." `http://www.omg.org/spec/BPMN/2.0/PDF`, Jan. 2011.

[6] A. Alves, A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, Sterling, D. König, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu, "Web Services Business Process Execution Language Version 2.0." OASIS Committee, 2007.

[7] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, "Web Services Choreography Description Language Version 1.0." World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109, 2005.

[8] M. P. Papazoglou, *Web Services - Principles and Technology*. Prentice Hall, 2008.

[9] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, pp. 7–18, 2010. 10.1007/s13174-010-0007-6.

[10] H. Jin, S. Ibrahim, T. Bell, W. Gao, D. Huang, and S. Wu, "Cloud types and services," in *Handbook of Cloud Computing* (B. Furht and A. Escalante, eds.), pp. 335–355, Springer US, 2010.

[11] E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good, "The cost of doing science on the cloud: the montage example," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 50:1–50:12, IEEE Press, 2008.

[12] M. G. Nanda, S. Chandra, and V. Sarkar, "Decentralizing composite web services." online, 2002.

[13] M. G. Nanda and N. Karnik, "Synchronization analysis for decentralizing composite web services," in *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, (New York, NY, USA), pp. 407–414, ACM, 2003.

[14] M. G. Nanda, S. Chandra, and V. Sarkar, "Decentralizing execution of composite web services," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004* (J. M. Vlissides and D. C. Schmidt, eds.), (Vancouver, BC, Canada), pp. 170–187, ACM, 2004.

[15] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda, "Decentralized orchestration of composite web services," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, (New York, NY, USA), pp. 134–143, ACM, 2004.

[16] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda, "Orchestrating composite web services under data flow constraints," in *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, (Washington, DC, USA), pp. 211–218, IEEE Computer Society, 2005.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.

[18] R. Khalaf and F. Leymann, "E role-based decomposition of business processes using bpel," in *Proceedings of the IEEE International Conference on Web Services*, ICWS '06, (Washington, DC, USA), pp. 770–780, IEEE Computer Society, 2006.

[19] R. Khalaf, O. Kopp, and F. Leymann, "Maintaining data dependencies across bpel process fragments," in *Proceedings of the 5th international conference on Service-Oriented Computing*, ICSOC '07, (Berlin, Heidelberg), pp. 207–219, Springer-Verlag, 2007.

[20] O. Kopp, R. Khalaf, and F. Leymann, "Deriving explicit data links in ws-bpel processes," in *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*, SCC '08, (Washington, DC, USA), pp. 367–376, IEEE Computer Society, 2008.

[21] L. Baresi, A. Maurino, and S. Modafferi, "Towards distributed bpel orchestrations," *ECEASST*, vol. 3, 2006.

[22] W. Fdhila, U. Yildiz, and C. Godart, "A flexible approach for automatic process decentralization using dependency tables," in *ICWS*, pp. 847–855, 2009.

[23] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[24] H. Eertink, W. Janssen, P. Luttighuis, W. Teeuw, and C. Vissers, "A business process design language," in *FM99 Formal Methods*, Springer Berlin / Heidelberg, 1999.

[25] O. M. Group, "BPMN 2.0 by Example Version 1.0 (non-normative)." `http://www.omg.org/spec/BPMN/2.0/examples/PDF`, Jan. 2002.

[26] R. Seguel Perez, *Business protocol adaptors for flexible chain formation and enactment*. PhD thesis, Eindhoven University of Technology, 2012.

[27] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr. 1989.

[28] K. Jensen, "Coloured petri nets," in *Petri Nets: Central Models and Their Properties* (W. Brauer, W. Reisig, and G. Rozenberg, eds.), vol. 254 of *Lecture Notes in Computer Science*, pp. 248–299, Springer Berlin / Heidelberg, 1987. 10.1007/BFb0046842.

[29] O. Kopp, D. Martin, D. Wutke, and F. Leymann, "The difference between graph-based and block-structured business process modelling languages," *Enterprise Modelling and Information Systems Architectures*, vol. 4, no. 1, pp. 3–13, 2009.

[30] A. Rensink, I. Boneva, H. Kastenberg, and T. Staijen, "User manual for the groove tool set," 2011.